Under consideration for publication in Formal Aspects of Computing

A program analysis framework for *tccp* based on abstract interpretation

Marco Comini[1] and María-del-Mar Gallardo[2] and Laura Titolo[3] and Alicia Villanueva[4]

[1]DIMI, Università degli Studi di Udine, Italy. (e-mail: marco.comini@uniud.it)
[2]LCC, Universidad de Málaga, Spain. (e-mail: gallardo@lcc.uma.es)[1]
[3]National Institute of Aerospace, USA. (e-mail: laura.titolo@nianet.org)
[4]DSIC, Universitat Politècnica de València, Spain. (e-mail: villanue@dsic.upv.es)[2]

Abstract. The Timed Concurrent Constraint Language (*tccp*) is a timed extension of the concurrent constraint paradigm. *tccp* was defined to model reactive systems, where infinite behaviors arise naturally. In previous works, a semantic framework and abstract diagnosis method for the language have been defined.

On the basis of that semantic framework, this paper proposes an abstract semantics that, together with a widening operator, is suitable for the definition of different analyses for *tccp* programs. The abstract semantics is correct and can be represented as a finite graph where each node represents a hypothetical (abstract) computational step of the program. The widening operator allows us to guarantee the convergence of the abstract fixpoint computation.

Keywords: concurrent constraint paradigm; abstract interpretation; abstract semantics; widening operators

1. Introduction

The Concurrent Constraint Paradigm (*ccp*, [Sar93]) is a simple, logic model which is different from other (concurrent) programming paradigms mainly due to the notion of store-as-constraint that replaces the classical store-as-valuation model. It is based on an underlying constraint system that handles constraints on variables and deals with partial information. Within this family, [dBGM00] introduced the *Timed Concurrent Constraint Language* (*tccp*) by adding to the original *ccp* model the notion of time and the ability to capture the absence of information. With these features, one can specify naturally behaviors typical of reactive systems such as *timeouts* or *preemption* actions.

It is well-known that modeling and analyzing concurrent systems by hand can be an extremely hard task. Thus, the development of automatic formal methods is essential. The particular characteristics of *ccp* languages make such task even harder, since we have to deal with technical issues due to the infinite

[1] This author has been supported by the Andalusian Excellence Project P11-TIC-7659.
[2] This work has been partially supported by the EU (FEDER) and the Spanish MINECO under grants TIN 2015-69175-C4-1-R and TIN 2013-45732-C4-1-P and by Generalitat Valenciana PROMETEOII/2015/013.
Correspondence and offprint requests to: M. Comini, M. Mar Gallardo, L. Titolo and A. Villanueva

computations (natural to reactive systems), use of negative information (particular for timed *ccp* languages) and non-determinism.

One well established technique to develop semantic-based program analysis is abstract interpretation [CC77], which relies on the definition of a specific approximated abstract semantics that captures the information needed to perform the analysis. Typically, one defines an over-approximation of the concrete semantics that includes all possible traces of the system, possibly introducing nonexistent ones. This allows one to develop (correct) analysis of *universal* properties. However, it does not allow one to analyze *existential* properties, for instance to verify that there exists a suspension trace. In our proposal, we follow such approach starting from the concrete semantics for *tccp* defined in [CTV13]. This semantics addresses all thorniest difficulties of *tccp* (i.e., infinite computations, use of negative information and non-determinism). Indeed, it is fully abstract w.r.t. the behavior of *tccp*. Furthermore, it is condensed, i.e., it employs in the denotations the minimal amount of information that is needed (to distinguish different behaviors). Therefore, such semantics is particularly well-suited as the base to apply abstract interpretation techniques, which take great advantage from a bottom-up and condensed definition. To the best of our knowledge, this is the only bottom-up and condensed semantics which is fully abstract w.r.t. the *full tccp* language. The fully-abstract denotational semantics of [dBGM00] captures just finite computations and has a top-down definition thus it is not well-suited for our purposes.

In the sequel, we define a framework of over-approximated abstract semantics parametric w.r.t. an abstract constraint system. This allows us to recycle all of the huge work done for developing abstract domains for logic programs (such as groundness analysis). More interestingly, we can also make new analyses for reactive systems such as non-suspension analysis and universal (safety and liveness) properties.

The proposed framework follows two complementary abstraction techniques. On the one hand, since processes in *tccp* communicate and synchronize through a shared store where both the presence and absence of information are relevant for the system progress, we make use of two dual notions of approximations (over/under-approximated relations) [AGPV05]. The combination of these two abstract relations allows us to guarantee the correctness of the abstraction and, at the same time, to do not lose too much precision.

On the other hand, since we need to preserve the notion of time—to be able to express properties of interest like safety or temporal properties—the abstract semantics domains that we need to consider are not Noetherian (even if we use finite abstract constraint systems). Thus, in order to have an effective technique, we use the widening approach of [BHRZ05, CC77] to ensure finiteness of the analysis. Given a *tccp* program, any instance of the abstraction framework computes an abstract graph that can be used to check relevant (temporal) system properties.

Applicability of our approach is illustrated by showing different analyses over our guiding example, a lift/passenger system. Thanks to the compositionality of the abstract semantics, we can focus the analysis on a particular process. For instance, we show that we can analyze properties regarding the lift process, independently from the rest of the system. More specifically, properties such as *the lift direction and floor are consistently updated* or *the lift never suspends* depend only on the lift process, thus we do not need to compute the semantics for the rest of the system. We can also check properties depending on the interaction among processes by considering the whole system. To this end, we describe how the abstract graph of the whole system lift/passenger can be constructed, and exemplify some properties that can be directly proved on the graph.

Due to abstract interpretation, in the abstract semantics we inevitably have spurious behaviors that can prevent us from proving specific properties. We discuss how properties such as *if the lift is going up, then it eventually will go down* could also be analyzed by applying techniques to remove spurious behaviors.

Contributions of this work. This paper is an extended version of [CGTV15], where a framework of abstract semantics suitable for program analysis of *tccp* programs was presented. In this paper, we present an improved version of the framework which admits richer abstract domains, and this allows us to handle more elaborated properties and systems. More specifically, this paper includes

- an abstract semantic domain schema which imposes less demanding conditions (w.r.t. [CGTV15]) on the relation between the abstract and the concrete constraint system underlying the language;
- results and proofs for the correctness of the proposed abstract semantics;
- a widening for the abstract semantics (already presented in [CGTV15]) that allows to effectively perform the analyses; and
- examples that illustrate both, the kind of properties that can be checked by means of the proposed

framework, and how the new schema allows us to use a more elaborated abstraction for the concrete constraint system.

Plan of the paper. Section 2 introduces the *tccp* language and the denotational model of the concrete semantics which is the basis for the definition of the abstract semantics. A guiding example is also introduced. Section 3 presents the proposed abstract semantics and the defined widening, which ensures finiteness. Section 4 proposes some specific analysis that can be defined on the proposed abstract semantics. Section 5 compares our proposal to related work and Section 6 concludes.

Proofs of correctness results can be consulted in Appendix A.

2. The *tccp* language

The *tccp* language [dBGM00] is particularly suitable to specify both reactive and time critical systems. As the other languages of the *ccp* paradigm [Sar93], it is parametric w.r.t. a *cylindric constraint system* which handles the data information of the program in terms of constraints. The computation progresses as the concurrent and asynchronous activity of several agents that can accumulate information in a store, or query information from it. A cylindric constraint system[3] is an algebraic structure $\langle \mathbf{C}, \leq, \otimes, \mathit{false}, \mathit{true}, \mathit{Var}, \exists \rangle$ composed of a set of constraints $\mathbf{C}$ such that $(\mathbf{C}, \leq)$ is a (complete) algebraic lattice where $\otimes$ is the *lub* operator and *false* and *true* are respectively the greatest and the least element of $\mathbf{C}$; *Var* is a denumerable set of variables and $\exists$ existentially quantifies variables over constraints. The *entailment* $\vdash$ is the inverse of $\leq$.

Given a cylindric constraint system $\mathbf{C}$ and a set of process symbols Π, the syntax of agents is given by the grammar

$$A ::= \mathsf{skip} \mid \mathsf{tell}(c) \mid A \parallel A \mid \exists x\, A \mid \textstyle\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A \mid \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ A \mid p(\vec{x})$$

where c, $c_1, \ldots, c_n$ are finite[4] constraints in $\mathbf{C}$; $p_{/m} \in \Pi$, and $\vec{x}$ denotes a generic tuple of m variables. A *tccp* program is an object of the form $D\,.\,A$, where A is an agent, called *initial agent*, and D is a set of *process declarations* of the form $p(\vec{x}) :\!- A$ (for some agent A). The notion of time is introduced by defining a discrete and global clock.

The *operational semantics* of *tccp*, defined in [dBGM00], is formally described by a transition system $T = (\mathit{Conf}, \longrightarrow)$ where configurations *Conf* are pairs $\langle A, c \rangle$ representing the agent A to be executed in the current global *store* $c \in \mathbf{C}$. Informally, the $\mathsf{tell}(c)$ agent adds the constraint c to the store in the next time instant and then stops. The choice agent $\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$ consults the store and non-deterministically executes (at the following time instant) one of the agents A_i whose corresponding guard c_i is entailed by the current store; otherwise, if no guard is entailed by the store, the agent suspends. The conditional agent $\mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B$ behaves in the current time instant like A (respectively B) if c is (respectively is not) entailed by the store. $A \parallel B$ models the parallel composition of A and B in terms of maximal parallelism. The agent $\exists x\, A$ makes variable x local to A. To this end, it uses the $\exists$ operator of the constraint system. Finally, the agent $p(\vec{x})$ takes non-deterministically from D a declaration of the form $p(\vec{x}) :\!- A$ and then executes A at the following time instant.

Example 2.1 (Guiding example) The following code shows a possible *tccp* implementation of a simple lift/passenger system. We assume that the lift is located at a building with $N+1$ floors numbered as $0, 1, \cdots, N$. The lift process uses variables to store the current floor where the lift is placed and the movement direction (*up*/*down*), respectively. At each time instant, the lift moves, if possible, to the following floor, according to the current movement direction. When the lift reaches floors 0 or N, then it changes the movement direction.

Process *pssngr* models the behavior of a client that wants to take the lift to go from origin floor O to destination floor D. This process makes use of variable *St* to store the passenger's state: *wait*, when she is waiting for the lift, *in*, when she is inside the lift, and *out*, when she has arrived at the destination floor.

The underlying concrete Cylindric Constraint System is formed by taking equivalence classes, modulo

[3] See [dBGM00, Sar93] for more details on cylindric constraint systems, where traditionally, the *glb* is not explicitly defined.

[4] The notion of finite constraints was formally defined in [SRP91] and, in the context of algebraic constraint systems, is equivalent to the notion of compact element.

logical equivalence, of finite conjunctions of (dis)equalities over variables, constants $\{up, down, in, out, wait\}$ and numbers $\{0, \ldots, N\}$ plus two arithmetic increment and decrement operations over integers. In this specific case the instance of $\otimes$ is thus conjunction, while $\preceq$ is the opposite of logical implication and $\exists_x$ is the operation that removes all conjuncts referring to variable x after information has been propagated within a constraint (e.g., $\exists_x(x = y \wedge x = 3) = y = 3$). Moreover, due to the monotonicity of the store, we use streams (written in a list-fashion way) to simulate *imperative-style* variables ([dBGM00]). In our example, CF, Dir and St are streams.

$$
\begin{array}{l}
main(N, O, D) :- \exists\, CF, Dir, St \,\big(lift(N, CF, Dir) \parallel pssngr(CF, O, D, St) \parallel \\
\qquad\qquad \mathsf{tell}(CF = [0 \mid \_]) \parallel \mathsf{tell}(Dir = [up \mid \_]) \parallel \mathsf{tell}(St = [wait|\_])\big) \\
lift(N, CF, Dir) :- \exists\, CF^l, Dir^l, F \,\big(\mathsf{now}(Dir = [up \mid \_] \wedge CF = [N \mid \_]) \\
\quad \mathsf{then}\ (\mathsf{tell}(Dir = [up \mid Dir^l]) \parallel \mathsf{tell}(Dir^l = [down \mid \_]) \parallel lift(N, CF, Dir^l)) \\
\quad \mathsf{else\ now}\ (Dir = [up \mid \_]) \\
\qquad \mathsf{then}\ (\mathsf{tell}(CF = [F \mid CF^l]) \parallel \mathsf{ask}(true) \rightarrow (\mathsf{tell}(CF^l = [F + 1 \mid \_]) \parallel lift(N, CF^l, Dir))) \\
\qquad \mathsf{else\ now}\ (Dir = [down \mid \_] \wedge CF = [0 \mid \_]) \\
\qquad\quad \mathsf{then}\ (\mathsf{tell}(Dir = [down \mid Dir^l]) \parallel \mathsf{tell}(Dir^l = [up \mid \_]) \parallel lift(N, CF, Dir^l)) \\
\qquad\quad \mathsf{else}\ (\mathsf{tell}(CF = [F \mid CF^l]) \parallel \mathsf{ask}(true) \rightarrow (\mathsf{tell}(CF^l = [F - 1 \mid \_]) \parallel lift(N, CF^l, Dir))))\big) \\
pssngr(CF, O, D, St) :- \exists\, St', O', D' \,\big(\\
\quad \mathsf{ask}(CF = [D \mid \_] \wedge St = [in|\_]) \rightarrow (\mathsf{tell}(St = [in \mid St']) \parallel \mathsf{tell}(St' = [out \mid \_])) \\
\quad + \mathsf{ask}(CF = [O \mid \_] \wedge St = [wait|\_]) \rightarrow (\mathsf{tell}(St = [wait|St']) \parallel \mathsf{tell}(St' = [in \mid \_]) \parallel \\
\qquad\qquad \mathsf{tell}(CF = [\_ \mid CF']) \parallel pssngr(CF', O, D, St')) \\
\quad + \mathsf{ask}((CF = [O' \mid \_] \wedge O' \neq O \wedge CF = [D' \mid \_] \wedge D' \neq D)) \rightarrow \\
\qquad\qquad (\mathsf{tell}(CF = [\_ \mid CF']) \parallel pssngr(CF', O, D, St'))\big) \\
\quad + \mathsf{ask}((CF = [D \mid \_] \wedge St \neq [in|\_])) \rightarrow (\mathsf{tell}(CF = [\_ \mid CF']) \parallel pssngr(CF', O, D, St'))\big) \\
\quad + \mathsf{ask}((CF = [O \mid \_] \wedge St \neq [wait|\_])) \rightarrow (\mathsf{tell}(CF = [\_ \mid CF']) \parallel pssngr(CF', O, D, St'))\big)
\end{array}
$$

Note that the three last possible branches have the same behavior, namely they perform a recursive call to the *pssngr* process with the updated argument values when the passenger state (*wait*, *in* or *out*) does not change. ■

2.1. The concrete denotational semantics

In this section, we briefly recall the concrete denotational domain and semantics of [CTV13], which is fully abstract[5] w.r.t. the small-step operational behavior of *tccp*. Fully detailed version of all definitions and proof of full abstraction, as well as of all results here summarized can be found in [CTV13].

Such semantics consists of a set of *conditional (timed) traces* that represent, in a compact way, all the possible behaviors that the program can manifest when executed with a specific *input* (initial store). Conditional traces can be seen as hypothetical computations in which, for each time instant, we have a condition representing the information that the global store has to satisfy in order to proceed to the next time instant. Briefly, a conditional trace is a (possibly infinite) sequence $t_1 \cdots t_n \cdots$ of *conditional states*, which can be of three forms:

conditional store: a pair $\eta \rightarrowtail c$, where η is a *condition*, defined below, and $c \in \mathbf{C}$ a store;
stuttering: the construct $stutt(C)$, with $C \subseteq \mathbf{C} \setminus \{true\}$;
end of a process: the construct $\boxtimes$, which cannot be followed by other conditional states.

The empty sequence of conditional states is denoted by ϵ. Intuitively, the conditional store $\eta \rightarrowtail c$ means that, if condition η is satisfied by the current store, the computation proceeds so that, in the following time

[5] Recall that fully abstract means that the semantics of two programs is identical if and only if the two programs have the same execution behavior.

instant, the store is c. The stuttering construct $stutt(\{c_1, \ldots, c_n\})$ models the suspension of the computation when none of the guards in an $\sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i$ is satisfied.

A *condition* η is a pair $\eta = (\eta^+, \eta^-)$ where $\eta^+ \in \mathbf{C}$ and $\eta^- \in \wp(\mathbf{C})$ are called positive and negative condition, respectively. The positive/negative condition represents information that a given store must/must not entail, thus they have to be consistent in the sense that $\forall c^- \in \eta^-, \eta^+ \nvdash c^-$. For instance, the condition $(x > 2, \{x > 1\})$ is not consistent since $x > 2 \vdash x > 1$. We also say that a store $c \in \mathbf{C}$ is *consistent* with η, written $c \gg \eta$, if $\eta^+ \otimes c \neq \mathit{false}$ and $\forall h \in \eta^-. c \nvdash h$. Moreover, we say that c *satisfies* η, written $c \models \eta$, when $c \vdash \eta^+$ and $\forall h \in \eta^-. c \nvdash h$.

Conditional traces are monotone (i.e., for each $t_i = \eta_i \rightarrowtail c_i$ and $t_j = \eta_j \rightarrowtail c_j$ such that $j \geq i$, $c_j \vdash c_i$) and consistent (i.e., each store in a trace does not entail the negative conditions of the following conditional state). $\mathbf{CT}$ is the set of all maximal conditional traces, i.e., infinite traces or finite traces ending with $\boxtimes$.

Example 2.2 (Conditional traces) It is easy to see that the sequence $r_1 \coloneqq (\mathit{true}, \varnothing) \rightarrowtail y = 0 \cdot (x > 2, \varnothing) \rightarrowtail y = 0 \otimes z = 3 \cdot \boxtimes$ is a conditional trace (composed by three conditional states) that satisfies monotonicity and consistency. On the contrary, $r_1' \coloneqq (\mathit{true}, \varnothing) \rightarrowtail y = 0 \cdot (x > 2, \{y \geq 0\}) \rightarrowtail y = 0 \otimes z = 3 \cdot \boxtimes$ is not consistent since the store of the first conditional state entails the only element in the negative condition of the successive conditional state, i.e., $y = 0 \vdash y \geq 0$. ■

Maximal conditional traces can be ordered, by structural induction, as follows: $\forall r \in \mathbf{CT}. \epsilon \sqsubseteq r, \boxtimes \sqsubseteq \boxtimes$, and

$$(\eta_1^+, \eta_1^-) \rightarrowtail c \cdot r_1 \sqsubseteq (\eta_2^+, \eta_2^-) \rightarrowtail c \cdot r_2 \iff \eta_1^+ \vdash \eta_2^+ \wedge \eta_2^- \trianglelefteq \eta_1^- \wedge r_1 \sqsubseteq r_2$$

$$stutt(\eta_1^-) \cdot r_1 \sqsubseteq stutt(\eta_2^-) \cdot r_2 \iff \eta_2^- \trianglelefteq \eta_1^- \wedge r_1 \sqsubseteq r_2$$

where $C \trianglelefteq C' \iff \forall c \in C. \exists c' \in C'. c \vdash c'$ [6]. Intuitively, a trace r is smaller than another trace r' if and only if the conditions of r are more (or equally) restrictive than those of r'. The intuition behind $C \trianglelefteq C'$ is that C' contains restrictions for the behaviors it represents that are not stronger than those in C. In other words, C won't discard more behaviors than C' when used as the negative condition of a conditional trace. For instance, $\{x > 20\} \trianglelefteq \{x > 10\}$, but $\{x > 20, y > 0\} \not\trianglelefteq \{x > 20\}$ since, due to the constraint $y > 0$, a conditional state with the latter set as negative condition might admit behaviors not admitted by one using the former set.

The order defined between maximal traces can be extended over sets $M_1, M_2 \subseteq \mathbf{CT}$ as $M_1 \sqsubseteq M_2 \iff \forall r_1 \in M_1 \exists r_2 \in M_2. r_1 \sqsubseteq r_2$. This relation induces the equivalence relation $M_1 \equiv M_2 \iff M_1 \sqsubseteq M_2 \wedge M_2 \sqsubseteq M_1$. We abuse notation and denote the quotient of $\sqsubseteq$ over equivalence classes with the same symbol. In the following, we use non-empty *maximal conditional trace sets* modulo $\equiv$ and denote their class by $\mathbb{C}$. $(\mathbb{C}, \sqsubseteq, \bigsqcup, \bigsqcap, [\mathbf{CT}]_{\equiv}, \{\epsilon\})$ is a complete lattice, where $M_1 \sqcup M_2$ is the equivalence class of set union and $M_1 \sqcap M_2$ is the set of maximal conditional traces such that each trace is less or equal than both a trace in M_1 and a trace in M_2 (i.e., it represents the intersection of the behaviors represented by M_1 and M_2).

The concrete semantics built on domain $\mathbb{C}$ is based on a semantics evaluation function $\mathcal{A}[\![A]\!]_{\mathcal{I}}$ which, given an agent A and an interpretation $\mathcal{I}$, builds the conditional traces associated to A. Such concrete denotational semantics is the basis of the abstract denotational semantics in Section 3, which is actually obtained just by replacing concrete semantics operations by their corresponding abstract versions (and thus the structure of their definition is the same). The interpretation $\mathcal{I}$ is a function which associates to each process symbol a set of maximal conditional traces "modulo variance".

Definition 2.3 (Interpretations) *Let* $\mathbb{PC} \coloneqq \{p(\vec{x}) \mid p \in \Pi$ *and* $\vec{x}$ *are distinct variables*$\}$. *An* interpretation *is a function* $\mathcal{I}: \mathbb{PC} \to \mathbb{C}$ *modulo variance*[7]. *The semantic domain* $\mathbb{I}$ *is the set of all interpretations ordered by the point-wise extension of* $\sqsubseteq$.

The semantics for a set of *tccp* process declarations D is the fixpoint $\mathcal{F}[\![D]\!] \coloneqq lfp(\mathcal{D}[\![D]\!])$ of the continuous operator $\mathcal{D}[\![D]\!]_{\mathcal{I}}(p(\vec{x})) \coloneqq \bigsqcup_{p(\vec{x}) :- A \in D} \mathcal{A}[\![A]\!]_{\mathcal{I}}$.

Example 2.4 (Semantics of our guiding example) The semantics of the *lift* process defined in Example 2.1 is graphically represented in Figure 1. Each branch of the tree corresponds to one of the branches of

[6] The $\trianglelefteq$ relation induces an equivalence relation on negative conditions (formally, $C \approx C' \iff C \trianglelefteq C' \wedge C' \trianglelefteq C$) and in the sequel we implicitly consider negative conditions modulo such equivalence.

[7] Two functions $I, J: \mathbb{PC} \to \mathbb{C}$ are *variants*, denoted by $I \cong J$, if for each $\pi \in \mathbb{PC}$ there exists a variable renaming ρ such that $(I(\pi))\rho = J(\pi\rho)$.

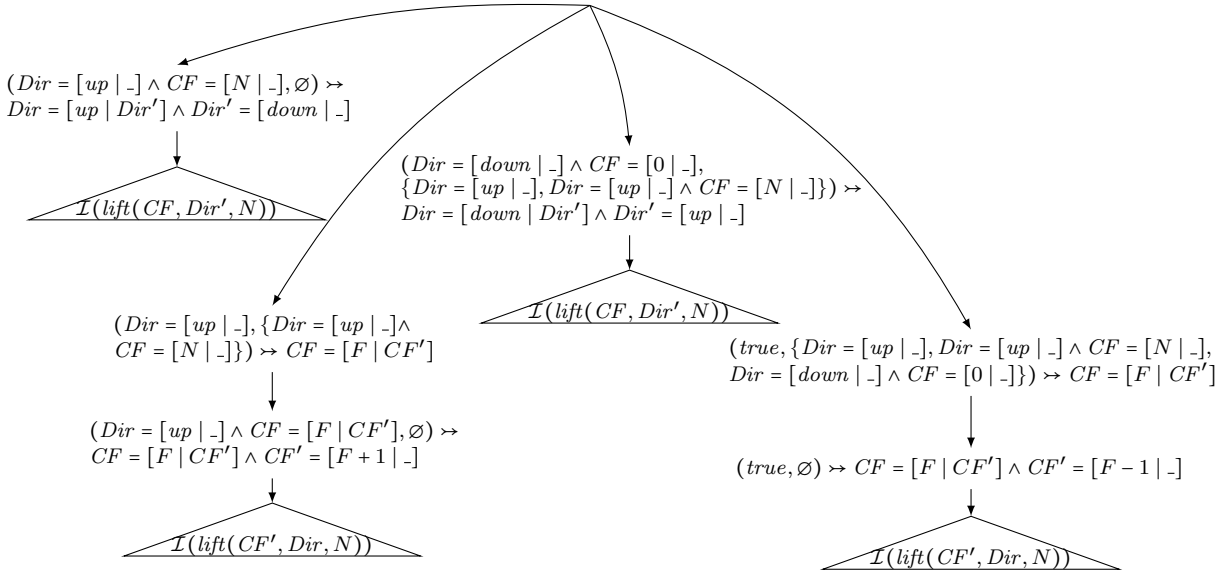


Figure 1. Representation of the semantics of the *lift* process.

the nested now agents. The first branch (in left-to-right order) represents the case in which the direction of the lift is *up* and the current floor is the last one (N). The second branch is taken when the direction is *up* but the current floor is not N (see the negative condition). In that case, the current floor changes from F to $F+1$. The third branch represents the case when the direction of the lift is *down* and the current floor is 0, thus the direction is changed to *up* by adding the constraint $Dir' = [up \mid \_]$. Finally, the fourth branch is taken when all the guards are not entailed (see the negative condition, composed by all the guards of the nested now agents). In that case, the lift moves to the lower floor $F-1$. In all the aforementioned cases, a recursive call is invoked appropriately. These calls are represented in Figure 1 by the triangles labeled with the interpretation of the process *lift*. ■

3. The (finite) abstract semantics for *tccp*

In this section, we define our over-approximated abstract semantics framework for *tccp*. It is parametric w.r.t. a Galois insertion $(\mathbf{C}, \vdash) \underset{\tau}{\overset{\tau^\gamma}{\leftrightarrows}} (\hat{\mathbf{C}}, \hat{\vdash})$ onto the *abstract constraint system* $\langle \hat{\mathbf{C}}, \hat{\leq}, \hat{\otimes}, \hat{false}, \hat{true}, Var, \hat{\exists} \rangle$. As usual, $\hat{true}$ and $\hat{false}$ are the smallest and the greatest abstract constraint, respectively. Moreover $\hat{\vdash}$ (called *abstract entailment*) is the inverse relation of $\hat{\leq}$. The *abstraction* function τ replaces exact (concrete) constraints of $\mathbf{C}$ by approximated (abstract) constraints of $\hat{\mathbf{C}}$ and preserves the concrete order $\vdash$ (which is associated to information content) with respect to the corresponding abstract order $\hat{\vdash}$. The *concretization* function τ^γ associates to each abstract constraint (of $\hat{\mathbf{C}}$) the maximal concrete constraint (of $\mathbf{C}$) that is approximated by it.

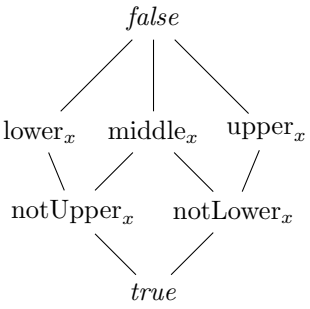
We illustrate the abstraction of constraint systems with two examples. The first one is the classical sign abstraction. The following one is used in Section 4 for the analysis of our guiding example.

Example 3.1 (Sign abstraction) Given the standard constraint system with inequalities over integer numbers, we abstract it to the abstract constraint system that contains only information about the sign of the system variables.

We define the abstract constraint system as $\langle \hat{\mathbf{S}}, \Leftarrow, \wedge, false, true, Var, \hat{\exists} \rangle$ where $\hat{\mathbf{S}}$ is the set of finite conjunctions of $\{\mathrm{pos}_x, \mathrm{neg}_x, \mathrm{zero}_x \mid x \in Var\} \cup \{false, true\}$ and $\hat{\exists}_x$ is the operation that deletes all atoms referring to variable x.

The abstract approximation τ is defined by cases as follows:

$$\tau(true) = true \qquad \tau(false) = false \qquad \tau(c \otimes c') = \tau(c) \wedge \tau(c') \qquad \tau(\exists_x c) = \hat{\exists}_x \tau(c)$$



$x = [F\|\_]$	$y = [F+1\|x]$	$y = [F-1\|x]$
lower$_x$	middle$_y$	*false*
middle$_x$	notLower$_y$	notUpper$_y$
upper$_x$	*false*	middle$_y$
notLower$_x$	notLower$_y$	notUpper$_y$
notUpper$_x$	notLower$_y$	notUpper$_y$
true	*true*	*true*

Figure 2. Floors lattice and abstract operations

$$\tau(x \le a) = \begin{cases} \text{neg}_x & \text{if } a \le 0 \\ \mathit{true} & \text{if } a > 0 \end{cases} \qquad \tau(x \ge a) = \begin{cases} \text{pos}_x & \text{if } a \ge 0 \\ \mathit{true} & \text{if } a < 0 \end{cases} \qquad \tau(x = a) = \begin{cases} \text{pos}_x & \text{if } a > 0 \\ \text{neg}_x & \text{if } a < 0 \\ \text{zero}_x & \text{if } a = 0 \end{cases}$$

■

Example 3.2 (Floors abstraction) The concrete constraint system of Example 2.1 can be approximated by the abstract constraint system $\langle \mathbf{F}, \Leftarrow, \wedge, \mathit{false}, \mathit{true}, \mathit{Var}, \tilde{\exists} \rangle$ where $\mathbf{F}$ is the set of finite conjunctions of $\{\text{middle}_x, \text{lower}_x, \text{upper}_x, \text{notLower}_x, \text{notUpper}_x \mid x \in \mathit{Var}\} \cup \{\mathit{false}, \mathit{true}\}$, ordered as depicted in the Hasse diagram of Figure 2 and $\tilde{\exists}_x$ is the operation that deletes all atoms referring to variable x. The table in the figure shows the definition for the increment and decrement operations in this abstract domain. These are the only two operations defined in the constraint system (and used in the program).

The abstraction of a stream constraint $\tau(X = [v|Y])$ approximates the numeric value v following the natural meaning of constants in the lattice $\mathbf{F}$, that is, lower$_x$/upper$_x$ for $v = 0/N$ and middle$_x$ when $0 < v < N$. Constants notLower$_x$/notUpper$_x$ (meaning $0 < v \le N$ and $0 \le v < N$) are used to abstract (more) precisely (different) values coming from non-deterministic computations. ■

The problem of abstracting constraint systems in the *ccp* paradigm was studied in [FGMP93, ZGL97], where abstraction meant loss of completeness but not of correctness. However, for the *tccp* case, due to the strong synchronization of parallel processes, over-approximation of stores can lead to lose correctness [AGPV05]. Hence, similarly to [AGPV05, CTV11], given an abstract domain, we use two binary relations (an over- and an under-approximation relation) over the abstract domain to be able to *conservatively* approximate the operational behavior. For positive conditions we need to guarantee to preserve all consistent constraints (thus over-approximate) while dually, for negative conditions we cannot introduce solutions but possibly discard some (thus under-approximate). Thanks to this combination, we do not lose concrete traces in the abstraction process and so we guarantee correctness of the abstract semantics.

The over-approximation $\rightarrowtail^+$ between abstract constraints can be defined systematically in terms of $\hat{\otimes}$ as $\forall \hat{a}, \hat{b} \in \hat{\mathbf{C}},\ \hat{a} \rightarrowtail^+ \hat{b} \Leftrightarrow \hat{a} \hat{\otimes} \hat{b} \neq \hat{\mathit{false}}$. This is closely related to the notion of relative-pseudo-complement. Intuitively, over-approximation holds if there exists the possibility that the entailment holds in the concrete domain. Namely, $\hat{a} \rightarrowtail^+ \hat{b}$ if $\tau^\gamma(\hat{a}) \otimes \tau^\gamma(\hat{b}) \neq \mathit{false}$.[8] Note that this is not equivalent to the abstract entailment ($\hat{\vdash}$) in $\hat{\mathbf{C}}$. Relation $\rightarrowtail^+$ is, in general, neither transitive nor reflexive.

On the contrary, for the under-approximation $\rightarrowtail^-$ we need to guarantee that, if it holds, then it also holds in the concrete. Namely, given two abstract constraints $\hat{a}, \hat{b} \in \hat{\mathbf{C}}$, $\hat{a} \rightarrowtail^- \hat{b} \iff \forall a, b \in \mathbf{C}. a \vdash \tau^\gamma(\hat{a}), b \vdash \tau^\gamma(\hat{b}) \Rightarrow a \vdash b$. Note that, in general, relation $\rightarrowtail^-$ is neither reflexive nor symmetric.

We use this relation in order to achieve better precision of the abstract semantics when handling negative information, i.e., to discard traces that do not correspond with real ones. Obviously, this is a very demanding notion. However, we have to guarantee correctness, thus we cannot discard a path to construct an abstract trace if there exists a single possibility (a single concretization) that follows such path. Hence, in general, we cannot use a less demanding notion.

[8] Recall that, the concretization function τ^γ is a part of the Galois insertion between constraint systems and that it is the adjoint of the abstraction function.

Example 3.3 (Approximation relations for Example 3.1) For the sign abstraction, we have that $\mathrm{pos}_x \not\succ^+ \mathrm{neg}_x$ since $\tau^\gamma(\mathrm{pos}_x) \otimes \tau^\gamma(\mathrm{neg}_x) = x = 0$. For instance, if a positive value (≥ 0) for x is required (by a condition) in the concrete domain (like $x = 0$), and the current abstract store is neg_x, the over-relation accepts neg_x since x could be 0.

The under-approximation is $\hat{a} \succ^- \hat{b} \Leftrightarrow (\hat{a} = \mathit{false}) \vee (\hat{a} = \mathrm{zero}_x \wedge \hat{b} = \mathrm{zero}_x)$. As already mentioned, in contrast to $\succ^+$, relation $\succ^-$ is used to discard traces. For instance, given the $\mathsf{now}\ x = 0\ \mathsf{then}\ A_1\ \mathsf{else}\ A_2$ agent, if the current abstract store is zero_x and thanks to this under-approximation, we are able to detect that the value of x cannot be different from zero and then our semantics does not include spurious traces representing the else branch. ■

Example 3.4 (Approximation relations for Example 3.2) By observing Figure 2, we see, for instance, that $\mathrm{middle}_x \not\succ^+ \mathrm{upper}_x$ and $\mathrm{lower}_x \not\succ^+ \mathrm{notLower}_x$, but $\mathrm{notLower}_x \succ^+ \mathit{notUpper}_x$ and $\mathit{middle}_x \succ^+ \mathit{notLower}_x$. We also observe clearly that the relation is not transitive. For instance, $\mathit{lower}_x \succ^+ \mathit{notUpper}_x$, $\mathit{notUpper}_x \succ^+ \mathit{middle}_x$ but $\mathit{lower}_x \not\succ^+ \mathit{middle}_x$.

For this example, the under-approximation relation is $\hat{a} \succ^- \hat{b} \Leftrightarrow (\hat{a} = \mathit{false}) \vee (\hat{a} = \mathrm{lower}_x \wedge \hat{b} = \mathrm{lower}_x) \vee (\hat{a} = \mathrm{upper}_x \wedge \hat{b} = \mathrm{upper}_x)$. Observe that, in this case, relation $\succ^-$ is not reflexive. For instance, $\mathit{middle}_x \not\succ^- \mathit{middle}_x$ since constraint middle_x is too imprecise to discard traces. For example, assume that the system asks whether the current floor CF is the second one, and the abstract store contains value middle_{CF}. It is clear that we cannot exactly know the current lift floor and, in consequence, we have to take into account both possibilities, i.e., the case when CF is 2, and the case when it is not. ■

It is worth remarking that the abstract entailment relation $\hat{\vdash}$ and the over-relation $\succ^+$ are supported by different intuitions. Let us illustrate this by using the sign abstraction domain of Example 3.1. Whereas the first relation is the partial order of the lattice (for instance, we have that $\tau(x \geq 5) \not\hat{\vdash} \tau(x = 0)$), the second one checks, in some way, whether there exist two concretizations of the abstract constraints that are related by $\vdash$ (for instance, we have that $\tau(x \geq 5) \succ^+ \tau(x = 0)$, since $x = 0 \vdash \tau^\gamma(\tau(x \geq 5)))$.

3.1. The abstract semantic domain

The abstract denotational semantics $\mathbb{A}$ is formed by *abstract conditional traces*, which (essentially) are conditional traces (recalled in Section 2.1) where conditions and stores are formed by approximated constraints instead of concrete ones.

Definition 3.5 (Abstract conditions) *Let $\hat{\mathbf{C}}$ be an Abstract Cylindric Constraint System.* Abstract conditions *over $\hat{\mathbf{C}}$ are pairs $(\hat{\eta}, \check{\eta})$ where*

- $\hat{\eta} \in \hat{\mathbf{C}}$ *is called* abstract positive condition*,*
- $\check{\eta} \in \hat{\mathbf{NC}}_{\hat{\mathbf{C}}}$ *is called* abstract negative condition*, and*
- $\hat{\mathbf{NC}}_{\hat{\mathbf{C}}} := \wp(\hat{\mathbf{C}})/_{\approx}$*, where, for each pair $C, C' \subseteq \hat{\mathbf{C}}$, $C \approx C' \iff C \mathrel{\hat{\trianglelefteq}} C' \wedge C' \mathrel{\hat{\trianglelefteq}} C$ and $C \mathrel{\hat{\trianglelefteq}} C' \iff \forall c \in C.\, \exists c' \in C'.\, c \mathrel{\hat{\vdash}} c'$.*

We simply denote $\hat{\mathbf{NC}}_{\hat{\mathbf{C}}}$ by $\hat{\mathbf{NC}}$ when clear from the context. Moreover, we abuse notation and denote $\hat{\trianglelefteq}/_{\approx}$ simply by $\hat{\trianglelefteq}$. We also define $[C]_{\approx} \mathbin{\hat{\uplus}} [C']_{\approx} := [C \cup C']_{\approx}$.

The conjunction of two abstract conditions $\tilde{\eta}_1 = (\hat{\eta}_1, \check{\eta}_1)$ and $\tilde{\eta}_2 = (\hat{\eta}_2, \check{\eta}_2)$ is defined as $\tilde{\eta}_1 \mathbin{\hat{\otimes}} \tilde{\eta}_2 := (\hat{\eta}_1 \mathbin{\hat{\otimes}} \hat{\eta}_2, \check{\eta}_1 \mathbin{\hat{\uplus}} \check{\eta}_2)$.

An abstract condition $(\hat{\eta}, \check{\eta})$ is consistent *when $\hat{\eta} \neq \hat{\mathit{false}}$ and, moreover, $\forall \eta' \in \check{\eta}. \hat{\eta} \not\hat{\vdash} \eta'$. We denote $\hat{\Delta}_{\hat{\mathbf{C}}}$ the set of abstract consistent conditions.*

An abstract store $\hat{c} \in \hat{\mathbf{C}}$ is consistent *with $\tilde{\eta} = (\hat{\eta}, \check{\eta}) \in \hat{\Delta}_{\hat{\mathbf{C}}}$, written $\hat{c} \mathrel{\hat{\gg}} \tilde{\eta}$, if $\hat{c} \mathbin{\hat{\otimes}} \hat{\eta} \neq \hat{\mathit{false}}$ and $\forall \eta' \in \check{\eta}. \hat{c} \not\hat{\vdash} \eta'$. Moreover, we say that $\hat{c}$* satisfies *$\tilde{\eta}$, written $\hat{c} \mathrel{\hat{\models}} \tilde{\eta}$, when $\hat{c} \mathrel{\hat{\vdash}} \hat{\eta}$ and $\forall \eta' \in \check{\eta}, \hat{c} \not\hat{\vdash} \eta'$.*

We define the existential quantification on conditions as $\hat{\exists}_x \tilde{\eta} := (\hat{\exists}_x \hat{\eta}, \hat{\exists}_x \check{\eta})$[9]*.*

Finally, we define $\bar{\tau} : \wp(\mathbf{C}) \to \hat{\mathbf{NC}}$ as $\bar{\tau}(C) := [\{\tau(c) \mid c \in C\}]_{\approx}$.

[9] We abuse notation and, given a set of abstract constraints C, we write by $\hat{\exists}_x C$ the natural extension of $\hat{\exists}$ to sets.

Constraint abstractions τ and $\bar{\tau}$ enjoy some interesting properties. Namely, given $c \in \mathbf{C}$ and a (concrete) condition (η^+, η^-),

$$(\eta^+, \eta^-) \text{ is consistent} \Longrightarrow (\tau(\eta^+), \bar{\tau}(\eta^-)) \text{ is (abstractly) consistent} \tag{3.1}$$

$$c \gg (\eta^+, \eta^-) \Longrightarrow \tau(c) \mathrel{\hat{\gg}} (\tau(\eta^+), \bar{\tau}(\eta^-)) \tag{3.2}$$

$$c \models (\eta^+, \eta^-) \Longrightarrow \tau(c) \mathrel{\hat{\models}} (\tau(\eta^+), \bar{\tau}(\eta^-)) \tag{3.3}$$

Those properties follow directly from the definition of the concrete and abstract relations and from the Galois insertion between the concrete and the abstract constraint system.

Definition 3.6 (Abstract conditional traces) *An abstract conditional trace is a (possibly infinite) sequence $\hat{t}_1 \cdots \hat{t}_n \cdots$ of* abstract conditional states*, which can be of three forms.*

Abstract conditional store: *a pair $\tilde{\eta} \rightarrowtail \hat{c}$, where $\tilde{\eta} \in \hat{\Delta}_{\hat{\mathbf{C}}}$ and $\hat{c} \in \hat{\mathbf{C}}$;*

Abstract stuttering: *the construct $stutt(\check{\eta})$, with $\check{\eta} \in \hat{\mathbf{NC}}$;*

Abstract end of a process: *the construct $\boxtimes$, which cannot be followed by other abstract conditional states.*

The empty sequence of abstract conditional states is denoted by ϵ.

Abstract conditional traces must respect the following properties:

(monotonicity) *for each $\hat{t}_i = \tilde{\eta}_i \rightarrowtail \hat{c}_i$ and $\hat{t}_j = \tilde{\eta}_j \rightarrowtail \hat{c}_j$ such that $j \geq i$, $\hat{c}_j \mathrel{\hat{\vdash}} \hat{c}_i$ and*

(consistency) *for each $\hat{t}_i = \tilde{\eta}_i \rightarrowtail \hat{c}_i$ and $\hat{t}_{i+1} = (\hat{\eta}_{i+1}, \check{\eta}_{i+1}) \rightarrowtail \hat{c}_{i+1}$, $\forall \eta' \in \check{\eta}_{i+1}, \hat{c}_i \not\mathrel{\hat{\vdash}} \eta'$.*

Definition 3.7 (Abstract semantic domain) *We denote the set of all abstract conditional traces for the abstract constraint system $\hat{\mathbf{C}}$ by $\mathbf{AT}_{\hat{\mathbf{C}}}$, or simply $\mathbf{AT}$ when clear from the context. We (partially) order abstract conditional traces by their information content as $\forall r \in \mathbf{AT}. \epsilon \leq r$, $\boxtimes \leq \boxtimes$, and ($\forall \hat{c}, \hat{\eta}_1, \hat{\eta}_2 \in \hat{\mathbf{C}}$, $\forall \check{\eta}_1, \check{\eta}_2 \in \hat{\mathbf{NC}}$, $\forall r_1, r_2 \in \mathbf{AT}$)*

$$(\hat{\eta}_1, \check{\eta}_1) \rightarrowtail \hat{c}_1 \cdot r_1 \leq (\hat{\eta}_2, \check{\eta}_2) \rightarrowtail \hat{c}_2 \cdot r_2 \iff \hat{\eta}_1 \mathrel{\hat{\vdash}} \hat{\eta}_2 \wedge \check{\eta}_2 \mathrel{\hat{\trianglelefteq}} \check{\eta}_1 \wedge \hat{c}_1 \mathrel{\hat{\vdash}} \hat{c}_2 \wedge r_1 \leq r_2$$

$$stutt(\check{\eta}_1) \cdot r_1 \leq stutt(\check{\eta}_2) \cdot r_2 \iff \check{\eta}_2 \mathrel{\hat{\trianglelefteq}} \check{\eta}_1 \wedge r_1 \leq r_2$$

We extend $\leq$ to sets of abstract conditional traces A_1, A_2 as $A_1 \leq A_2 \iff \forall r_1 \in A_1 \exists r_2 \in A_2. r_1 \leq r_2$. Moreover, we define $A_1 \equiv A_2 \iff A_1 \leq A_2 \wedge A_2 \leq A_1$. In the sequel, we abuse notation and denote $\leq\!\!\!/_{\equiv}$ as $\leq$.

In the following, we use sets of non-empty abstract conditional traces modulo $\equiv$ and denote their class by $\mathbb{A}$. $(\mathbb{A}, \leq, \bigvee, \bigwedge, [\mathbf{AT}]_{\equiv}, \{\epsilon\})$ is a complete lattice, where $A_1 \vee A_2 = [A_1 \cup A_2]_{\equiv}$ and $A_1 \wedge A_2 = \left[\{r \in \mathbf{AT} \mid \exists r_1 \in A_1. r \leq r_1, \exists r_2 \in A_2. r \leq r_2\}\right]_{\equiv}$.

We can now define the abstraction of conditional traces.

Definition 3.8 (Conditional trace abstraction) *We define the abstraction function $\alpha^\tau : \mathbb{C} \to \mathbb{A}$ as the extension to sets (modulo $\equiv$) of function $\alpha^\tau : \mathbf{CT} \to \mathbf{AT}$ defined as follows. Given a conditional trace $t \in \mathbf{CT}$, $\alpha^\tau(\epsilon) = \epsilon$, $\alpha^\tau(\boxtimes) = \boxtimes$, $\alpha^\tau((\eta^+, \eta^-) \rightarrowtail c \cdot t') = (\tau(\eta^+), \bar{\tau}(\eta^-)) \rightarrowtail \tau(c) \cdot \alpha^\tau(t')$ and $\alpha^\tau(stutt(C) \cdot t') = stutt(\bar{\tau}(C)) \cdot \alpha^\tau(t')$. The concretization function γ^τ that, given a set of abstract traces, produces all the concrete traces that can be approximated by it, is defined as the adjoint of α^τ.*

For example, given the trace $r = stutt(\{X > 5\}) \cdot (X > 6, \{Y < 0\}) \rightarrowtail X > 9$ for the sign abstraction τ of Example 3.1 we have $\alpha^\tau(r) = stutt(\{\mathrm{pos}_X\}) \cdot (\mathrm{pos}_X, \{\mathrm{neg}_Y\}) \rightarrowtail \mathrm{pos}_X$.

3.2. The abstract semantics

The Galois insertion defined before can be naturally lifted to the domain of interpretations. We denote as $\mathbb{I}_{\mathbb{A}} := [\mathbb{P}\mathbb{C} \to \mathbb{A}]$ the abstract counterpart of $\mathbb{I}$.

The abstract semantics for a *tccp* program is based on the evaluation function for *tccp* agents shown in the following Definition 3.18. First, we need some auxiliary operators and properties. For the sake of readability, some (correctness) results together with their proofs can be consulted in the appendix.

To propagate information when composing traces, we use two propagation operators. The *abstract*

(strong) propagation operator $\hat{\Downarrow}$ is a partial function $\mathbf{AT} \times \hat{\mathbf{C}} \to \mathbf{AT}$ that instantiates an abstract conditional trace with a given abstract constraint and checks the consistency of the new information with the conditional states in the trace. This information needs to be propagated to all conditional states (including future states) in order to maintain the monotonicity of the store. Following these intuitions, operator $r\hat{\downarrow}_{\hat{c}}$ propagates $\hat{c}$ in the stores and conditions occurring in r, whereas the *abstract weak propagation operator* $r\hat{\downarrow}_{\hat{c}}$ propagates $\hat{c}$ only in the conditions.

Definition 3.9 (Abstract propagation operator) *The* abstract propagation *partial function* $\hat{\Downarrow} : \mathbf{AT} \times \hat{\mathbf{C}} \to \mathbf{AT}$ *is defined by structural induction as:* $\epsilon\hat{\Downarrow}_{\hat{c}} = \epsilon$, $\boxtimes\hat{\Downarrow}_{\hat{c}} = \boxtimes$ *and*

$$((\hat{\eta}, \check{\eta}) \rightarrowtail \hat{d} \cdot r')\hat{\Downarrow}_{\hat{c}} = \begin{cases} (\hat{\eta} \mathbin{\hat{\otimes}} \hat{c}, \check{\eta}) \rightarrowtail \hat{d} \mathbin{\hat{\otimes}} \hat{c} \cdot (r'\hat{\Downarrow}_{\hat{c}}) & \text{if } \hat{c} \mathbin{\hat{\gg}} (\hat{\eta}, \check{\eta}),\ \hat{d} \mathbin{\hat{\otimes}} \hat{c} \neq \hat{false} \\ (\hat{\eta} \mathbin{\hat{\otimes}} \hat{c}, \check{\eta}) \rightarrowtail \hat{false} \cdot \boxtimes & \text{if } \hat{c} \mathbin{\hat{\gg}} (\hat{\eta}, \check{\eta}),\ \hat{d} \mathbin{\hat{\otimes}} \hat{c} = \hat{false} \end{cases}$$

$$(stutt(\check{\eta}) \cdot r')\hat{\Downarrow}_{\hat{c}} = stutt(\check{\eta}) \cdot (r'\hat{\Downarrow}_{\hat{c}}) \quad \text{if } \forall \eta' \in \check{\eta}.\, \hat{c} \not\vdash \eta'$$

It is worth noting that if $\hat{\eta} \mathbin{\hat{\otimes}} \hat{c} = \hat{false}$, then the condition is not consistent, thus no conditional trace is produced.

Example 3.10 (Abstract propagation operator) Given $r = (\mathrm{pos}_x, \{\mathrm{zero}_y\}) \rightarrowtail \mathrm{pos}_y \cdot \boxtimes$, by definition, $r\hat{\downarrow}_{\mathrm{zero}_x} = (\mathrm{zero}_x, \{\mathrm{zero}_y\}) \rightarrowtail \mathrm{pos}_y \hat{\otimes} \mathrm{zero}_x \cdot \boxtimes$. On the contrary, $r\hat{\downarrow}_{\mathrm{zero}_y}$ is not defined since $\mathrm{zero}_y \not\gg (\mathrm{pos}_x, \{\mathrm{zero}_y\})$. ■

Definition 3.11 (Abstract weak propagation operator) *The* abstract weak propagation *partial function* $\hat{\downarrow} : \mathbf{AT} \times \hat{\mathbf{C}} \to \mathbf{AT}$ *is defined by structural induction as:* $\epsilon\hat{\downarrow}_{\hat{c}} = \epsilon$, $\boxtimes\hat{\downarrow}_{\hat{c}} = \boxtimes$ *and*

$$((\hat{\eta}, \check{\eta}) \rightarrowtail \hat{d} \cdot r')\hat{\downarrow}_{\hat{c}} = (\hat{c} \mathbin{\hat{\otimes}} \hat{\eta}, \check{\eta}) \rightarrowtail \hat{d} \cdot (r'\hat{\downarrow}_{\hat{c}}) \quad \text{if } \hat{c} \mathbin{\hat{\gg}} (\hat{\eta}, \check{\eta})$$

$$(stutt(\check{\eta}) \cdot r')\hat{\downarrow}_{\hat{c}} = stutt(\check{\eta}) \cdot (r'\hat{\downarrow}_{\hat{c}}) \quad \text{if } \forall \eta' \in \check{\eta}.\, \hat{c} \not\vdash \eta'$$

Analogously to $\hat{\Downarrow}$, note that if $\hat{\eta} \mathbin{\hat{\otimes}} \hat{c} = \hat{false}$, then the condition is not consistent, thus no conditional trace is produced by $r\hat{\downarrow}_{\hat{c}}$.

Example 3.12 (Abstract propagation operator) Given the same $r = (\mathrm{pos}_x, \{\mathrm{zero}_y\}) \rightarrowtail \mathrm{pos}_y \cdot \boxtimes$ as in the Example 3.10, $r\hat{\downarrow}_{\mathrm{zero}_x} = (\mathrm{zero}_x, \{\mathrm{zero}_y\}) \rightarrowtail \mathrm{pos}_y \cdot \boxtimes$. Also the weak propagation of zero_y $(r\hat{\downarrow}_{\mathrm{zero}_y})$ is not defined. ■

The $\hat{\|}$ operator composes two traces by consistently merging their conditions and stores. To this end, it uses the two propagation operators in order to merge the information from the traces. Information in the stores is (strongly) propagated, whereas information in the conditions is weakly propagated.

Definition 3.13 (Abstract parallel composition) *The* abstract parallel composition *partial operator* $\hat{\|}: \mathbf{AT} \times \mathbf{AT} \to \mathbf{AT}$ *is the commutative closure of the following partial operation defined by structural induction as:* $r \mathbin{\hat{\|}} \epsilon := r$, $r \mathbin{\hat{\|}} \boxtimes := r$ *and*

$$(stutt(\check{\eta}_1) \cdot r_1) \mathbin{\hat{\|}} (stutt(\check{\eta}_2) \cdot r_2) := stutt(\check{\eta}_1 \mathbin{\hat{\uplus}} \check{\eta}_2) \cdot (r_1 \mathbin{\hat{\|}} r_2)$$

Moreover, if $\tilde{\eta}_1 \mathbin{\hat{\otimes}} \tilde{\eta}_2$ *is consistent , then*

$$(\tilde{\eta}_1 \rightarrowtail \hat{c}_1 \cdot r_1) \mathbin{\hat{\|}} (\tilde{\eta}_2 \rightarrowtail \hat{c}_2 \cdot r_2) := \begin{cases} \tilde{\eta}_1 \mathbin{\hat{\otimes}} \tilde{\eta}_2 \rightarrowtail \hat{c}_1 \mathbin{\hat{\otimes}} \hat{c}_2 \cdot ((r_1\hat{\downarrow}_{\hat{\eta}_2}\hat{\Downarrow}_{\hat{c}_2}) \mathbin{\hat{\|}} (r_2\hat{\downarrow}_{\hat{\eta}_1}\hat{\Downarrow}_{\hat{c}_1})) & \text{if } \hat{c}_1 \mathbin{\hat{\otimes}} \hat{c}_2 \neq \hat{false} \\ \tilde{\eta}_1 \mathbin{\hat{\otimes}} \tilde{\eta}_2 \rightarrowtail \hat{false} \cdot \boxtimes & \text{if } \hat{c}_1 \mathbin{\hat{\otimes}} \hat{c}_2 = \hat{false}, \end{cases}$$

Finally, if $\forall \eta' \in \check{\eta}_2.\, \hat{\eta}_1 \not\vdash \eta'$, *then*

$$((\hat{\eta}_1, \check{\eta}_1) \rightarrowtail \hat{c}_1 \cdot r_1) \mathbin{\hat{\|}} (stutt(\check{\eta}_2) \cdot r_2) := (\hat{\eta}_1, \check{\eta}_1 \mathbin{\hat{\uplus}} \check{\eta}_2) \rightarrowtail \hat{c}_1 \cdot (r_1 \mathbin{\hat{\|}} (r_2\hat{\downarrow}_{\hat{\eta}_1}\hat{\Downarrow}_{\hat{c}_1}))$$

Clearly, by definition, $\hat{\|}$ is commutative. Moreover, because of $\hat{\otimes}$ associativity, $\hat{\|}$ is also associative. It is worth noting that if the propagated constraint is in contradiction with a condition of trace r, then the parallel composition is not defined on that r.

Example 3.14 (Parallel operator) Given $\hat{r}_1 = (\mathrm{pos}_x, \varnothing) \rightarrowtail \mathrm{pos}_y \cdot \boxtimes$ and $\hat{r}_2 = \mathit{stutt}(\{\mathrm{pos}_x, \mathrm{pos}_y\}) \cdot (\hat{\mathit{true}}, \varnothing) \rightarrowtail \mathrm{pos}_x \cdot \boxtimes$, the parallel composition $\hat{r}_1 \mathbin{\hat{\|}} \hat{r}_2 = (\mathrm{pos}_x, \{\mathrm{pos}_x, \mathrm{pos}_y\}) \rightarrowtail \mathrm{pos}_y \cdot (\mathrm{pos}_x \hat{\otimes} \mathrm{pos}_y, \varnothing) \rightarrowtail \mathrm{pos}_x \hat{\otimes} \mathrm{pos}_y \cdot \boxtimes$. Following the definition, the first conditional state corresponds to that of $\hat{r}_1$ updated with the negative condition of the stuttering conditional state. Then, the information in the first conditional state of $\hat{r}_1$ is propagated to the rest of conditional trace of $\hat{r}_2$. Note that since the second conditional state of $\hat{r}_2$ is the end-of-process mark, no other merges are needed.

We remark the fact that the initial condition in the resulting abstract conditional trace is consistent since $\mathrm{pos}_x \not\vdash \mathrm{pos}_x$. This corresponds to the idea of not discarding abstract traces unless we are sure that they do not admit concrete real behaviors. Let us illustrate this issue. Consider the concrete traces $r_1 = (x \geq 0, \varnothing) \rightarrowtail y = 1 \cdot \boxtimes$ and $r_2 = \mathit{stutt}(\{x \geq 2, y \geq 2\}) \cdot (\mathit{true}, \varnothing) \rightarrowtail x \geq 0 \cdot \boxtimes$ which are in the concretization of $\hat{r}_1$ and $\hat{r}_2$, respectively. Their (concrete) parallel composition is a consistent trace, i.e., $(x \geq 0, \{x \geq 2, y \geq 2\}) \rightarrowtail y = 1 \cdot (x \geq 0 \otimes y = 1, \varnothing) \rightarrowtail x \geq 0 \hat{\otimes} y = 1 \cdot \boxtimes$.

Let us now show the (natural) loss of precision of the abstract domain. Also $r_1' = (x = 1, \varnothing) \rightarrowtail y = 1 \cdot \boxtimes$ and $r_2' = \mathit{stutt}(\{x \geq 0, y \geq 0\}) \cdot (\mathit{true}, \varnothing) \rightarrowtail x \geq 0 \cdot \boxtimes$ are in the concretization of $\hat{r}_1$ and $\hat{r}_2$, respectively. However this time their (concrete) parallel composition is not defined because the first conditional state would be $(x = 1, \{x \geq 0, y \geq 0\}) \rightarrowtail y = 1$ but since $x = 1 \vdash x \geq 0$, the condition is not consistent. ■

The abstract *hiding operator* hides the information regarding some given variables in a trace.

Definition 3.15 (Abstract hiding operator) *The* abstract hiding operator *is the partial function* $\hat{\exists} : \wp(\mathit{Var}) \times \mathbf{AT} \to \mathbf{AT}$ *defined by structural induction as:*

$$\hat{\exists}_V\, r := \begin{cases} (\hat{\exists}_V\, \hat{\eta}, \hat{\exists}_V\, \check{\eta}) \rightarrowtail \hat{\exists}_V\, \hat{c} \cdot \hat{\exists}_V\, r' & \text{if } r = (\hat{\eta}, \check{\eta}) \rightarrowtail \hat{c} \cdot r' \\ \mathit{stutt}(\hat{\exists}_V\, \check{\eta}) \cdot \hat{\exists}_V\, r' & \text{if } r = \mathit{stutt}(\check{\eta}) \cdot r' \\ r & \text{if } r = \epsilon \text{ or } r = \boxtimes \end{cases}$$

where, for all $\hat{c} \in \hat{\mathbf{C}}$, $\hat{\exists}_{\{x_1,\ldots,x_n\}}\, \hat{c} := \hat{\exists}_{x_1} \cdots \hat{\exists}_{x_n}\, \hat{c}$ *and, for all* $C \in \hat{\mathbf{NC}}$, $\hat{\exists}_V\, C := \{\hat{\exists}_V\, \hat{c} \mid \hat{c} \in C\}$.

We abuse notation and write $\hat{\exists}_x\, r$ *for* $\hat{\exists}_{\{x\}}\, r$.

Definition 3.16 (Abstractly self-sufficient and x-self-sufficient conditional trace) *An abstract trace* $r \in \mathbf{AT}$ *is said to be* abstractly self-sufficient *if the first condition is* $(\hat{\mathit{true}}, \varnothing)$ *and, for each* $\hat{t}_i = (\hat{\eta}_i, \check{\eta}_i) \rightarrowtail \hat{c}_i$ *and* $\hat{t}_{i+1} = (\hat{\eta}_{i+1}, \check{\eta}_{i+1}) \rightarrowtail \hat{c}_{i+1}$, $\hat{c}_i \mathrel{\hat{\models}} \eta_{i+1}$. *In other words, each abstract store (abstractly) satisfies the successive abstract condition.*

Moreover, r *is* abstractly self-sufficient w.r.t. $x \in \mathit{Var}$ *(*x-self-sufficient*) if* $\hat{\exists}_{\mathit{Var} \setminus \{x\}}\, r$ *is self-sufficient. In other words, for abstractly self-sufficient conditional traces, no additional information (from other agents) is needed in order to* complete *the computation.*

Example 3.17 (Abstractly self-sufficient and x-self-sufficient conditional trace) Let us consider the abstract constraint system of Example 3.2. The abstract conditional trace $\hat{r} = (\hat{\mathit{true}}, \varnothing) \rightarrowtail \mathrm{pos}_x \cdot (\mathrm{zero}_x, \varnothing) \rightarrowtail \mathrm{zero}_x \hat{\otimes} \mathrm{pos}_y \cdot \boxtimes$ is not self-sufficient since $\mathrm{pos}_x \mathrel{\hat{\not\models}} (\mathrm{neg}_x, \varnothing)$.

Now consider a variation where we add the information zero_x to the stores, namely $\hat{r}' := (\mathit{true}, \varnothing) \rightarrowtail \mathrm{pos}_x \hat{\otimes} \mathrm{zero}_x \cdot (\mathrm{zero}_x, \varnothing) \rightarrowtail \mathrm{zero}_x \hat{\otimes} \mathrm{pos}_y \cdot \boxtimes$. It is easy to see that $\hat{r}'$ is a self-sufficient conditional trace, essentially because we add enough information in the first store to satisfy the second condition, i.e., $\mathrm{zero}_x \mathrel{\hat{\models}} (\mathrm{zero}_x, \varnothing)$.

Moreover, $\hat{r}'$ is also x-self-sufficient since $\hat{\exists}_{\mathit{Var} \setminus \{x\}}\, \hat{r}' = (\mathit{true}, \varnothing) \rightarrowtail \mathrm{pos}_x \hat{\otimes} \mathrm{zero}_x \cdot (\mathrm{zero}_x, \varnothing) \rightarrowtail \mathrm{zero}_x \cdot \boxtimes$, which is a self-sufficient trace. ■

Now we are ready to define the semantics evaluation function for agents, which is the core of the abstract semantics. As already said, this definition is structurally identical to the definition of the concrete evaluation function.

Definition 3.18 (Abstract Semantics Evaluation Function for Agents) *Given a tccp agent* A *and an (abstract) interpretation* $\mathcal{I}^\alpha \in \mathbb{I}_{\mathbb{A}}$, *we define by structural induction the* semantics evaluation $\mathcal{A}^\alpha[\![A]\!]_{\mathcal{I}^\alpha} \in \mathbb{A}$ *as follows.*

$$\mathcal{A}^\alpha[\![\mathsf{skip}]\!]_{\mathcal{I}^\alpha} := \{\boxtimes\} \tag{3.4a}$$

$$\mathcal{A}^\alpha[\![\mathsf{tell}(c)]\!]_{\mathcal{I}^\alpha} := \{(\hat{\mathit{true}}, \varnothing) \rightarrowtail \tau(c) \cdot \boxtimes\} \tag{3.4b}$$

$$\mathcal{A}^\alpha[\![A \parallel B]\!]_{\mathcal{I}^\alpha} \coloneqq \bigsqcup\{r_A \mathbin{\hat{\parallel}} r_B \mid r_A \in \mathcal{A}^\alpha[\![A]\!]_{\mathcal{I}^\alpha},\, r_B \in \mathcal{A}^\alpha[\![B]\!]_{\mathcal{I}^\alpha}\} \tag{3.4c}$$

$$\mathcal{A}^\alpha[\![\exists x\, A]\!]_{\mathcal{I}^\alpha} \coloneqq \bigsqcup\{\hat{\exists}_x\, r \mid r \in \mathcal{A}^\alpha[\![A]\!]_{\mathcal{I}^\alpha},\, r \textit{ is abstractly } x\textit{-self-sufficient}\} \tag{3.4d}$$

$$\mathcal{A}^\alpha[\![p(\overrightarrow{x})]\!]_{\mathcal{I}^\alpha} \coloneqq \{(\hat{true}, \varnothing) \rightarrowtail \hat{true} \cdot r \mid r \in \mathcal{I}^\alpha(p(\overrightarrow{x}))\} \tag{3.4e}$$

$$\mathcal{A}^\alpha[\![\sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i]\!]_{\mathcal{I}^\alpha} \coloneqq \bigsqcup\{\underbrace{stt \cdot \ldots \cdot stt}_{m} \cdot r \mid m \in \mathbb{N}, r \in M\} \sqcup \{stt \cdot \ldots \cdot stt \cdot \ldots\} \tag{3.4f}$$

where $stt \coloneqq stutt([\overline{\tau}(\{c_1, \ldots, c_n\})]_{\equiv})$ *and* $M = \bigsqcup\{(\tau(c_i), \varnothing) \rightarrowtail \hat{true} \cdot (r' \hat{\downarrow}_{\tau(c_i)}) \mid 1 \le i \le n,\, r' \in \mathcal{A}^\alpha[\![A_i]\!]_{\mathcal{I}^\alpha}\}$

$$\begin{aligned}
&\mathcal{A}^\alpha[\![\mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B]\!]_{\mathcal{I}^\alpha} \coloneqq \{(\tau(c), \varnothing) \rightarrowtail \hat{true} \cdot \boxtimes \mid \boxtimes \in \mathcal{A}[\![A]\!]_{\mathcal{I}^\alpha}\} \sqcup \\
&\quad \bigsqcup\{(\tau(c) \mathbin{\hat{\otimes}} \hat{\eta}, \check{\eta}) \rightarrowtail \hat{d} \cdot (r \hat{\downarrow}_{\tau(c)}) \mid (\hat{\eta}, \check{\eta}) \rightarrowtail \hat{d} \cdot r \in \mathcal{A}^\alpha[\![A]\!]_{\mathcal{I}^\alpha},\, \forall \eta' \in \check{\eta},\, \tau(c) \mathbin{\hat{\otimes}} \hat{\eta} \not\vdash \eta'\} \sqcup \\
&\quad \bigsqcup\{(\tau(c), \check{\eta}) \rightarrowtail \hat{true} \cdot r \hat{\downarrow}_{\tau(c)}) \mid stutt(\check{\eta}) \cdot r \in \mathcal{A}^\alpha[\![A]\!]_{\mathcal{I}^\alpha}, \forall \eta' \in \check{\eta},\, \tau(c) \not\vdash \eta'\} \sqcup \\
&\quad \bigsqcup\{(\hat{true}, \{\tau(c)\}) \rightarrowtail \hat{true} \cdot \boxtimes \mid \boxtimes \in \mathcal{A}^\alpha[\![B]\!]_{\mathcal{I}^\alpha}\} \sqcup \\
&\quad \bigsqcup\{(\hat{\eta}, \check{\eta} \mathbin{\hat{\uplus}} \{\tau(c)\}) \rightarrowtail \hat{d} \cdot r \mid\ (\hat{\eta}, \check{\eta}) \rightarrowtail \hat{d} \cdot r \in \mathcal{A}^\alpha[\![B]\!]_{\mathcal{I}^\alpha},\, \hat{\eta} \not\vdash \tau(c)\} \sqcup \\
&\quad \bigsqcup\{(\hat{true}, \check{\eta} \mathbin{\hat{\uplus}} \{\tau(c)\}) \rightarrowtail \hat{true} \cdot r \mid stutt(\check{\eta}) \cdot r \in \mathcal{A}^\alpha[\![B]\!]_{\mathcal{I}^\alpha}\}
\end{aligned} \tag{3.4g}$$

The semantics for a set of process declarations D *is the fixpoint* $\mathcal{F}^\alpha[\![D]\!] \coloneqq lfp(\mathcal{D}^\alpha[\![D]\!])$ *of the continuous operator* $\mathcal{D}^\alpha[\![D]\!]_{\mathcal{I}^\alpha}(p(\overrightarrow{x})) \coloneqq \bigvee_{p(\overrightarrow{x}) :- A \in D} \mathcal{A}^\alpha[\![A]\!]_{\mathcal{I}^\alpha}$.

We explain in detail some significant cases. The semantics of the $\mathsf{tell}(c)$ agent (3.4a) has a trace with two conditional states: the first one with condition $(\hat{true}, \varnothing)$, which is the less demanding condition since c must be added to the store in any case (in the next time instant). Next, the computation terminates with the end-of-process symbol $\boxtimes$. The parallel and hiding cases are defined in terms of the corresponding auxiliary operators, whereas the case of a process call $p(\overrightarrow{x})$ is the abstract behavior of the process specified by the interpretation $\mathcal{I}^\alpha$ (i.e., $r \in \mathcal{I}^\alpha(p(\overrightarrow{x}))$) but starting at the following time instant (since in *tccp* process calls introduce a delay of one time unit).

The case for the non-deterministic choice deserves special attention since, without the under-approximation relation, due to suspension, concrete behaviors might be lost, which would make the abstract semantics incomplete and the subsequent analysis unsound. Intuitively, we need to guarantee that, in the abstract semantics, we keep a conditional trace modeling suspension. The abstract conditional trace that models suspension will be discarded just in case that it becomes inconsistent when merged (by means of the parallel operator) with another (abstract) conditional trace. Thus, the semantics for the (non-deterministic) choice (3.4f) collects, for each guard c_i, a conditional trace of the form $(\tau(c_i), \varnothing) \rightarrowtail \hat{true} \cdot (r \hat{\downarrow}_{\tau(c_i)})$. This trace requires that $\tau(c_i)$ has to be satisfied by the current store. Then, the constraint $\tau(c_i)$ is propagated to the conditions in trace r (the continuation of the computation, which belongs to the semantics of A_i). Furthermore, we collect the stuttering traces, which correspond to the case when the computation suspends. These traces are of the form $stt \cdot \ldots \cdot stt \cdot r$, where r is one of the traces above, *plus* the infinite stuttering behavior $stt \cdot \ldots \cdot stt \cdot \ldots$.

The definition (3.4g) for the conditional agent $\mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B$ is similar to the previous case. However, since the now construct must be instantaneous, in order to correctly model the timing of the agent, we have six cases depending on the possible forms of the first conditional state of the semantics of A (respectively B) and on the fact that the guard c is satisfied or not in the current time instant.

Abstract semantic operators $\mathcal{A}^\alpha$ and $\mathcal{D}^\alpha$ are correct abstractions of $\mathcal{A}$ and $\mathcal{D}$, as stated in the theorem below. Hence, abstract interpretation theory ensures that $\mathcal{F}^\alpha[\![D]\!]$ is a correct approximation of $\mathcal{F}[\![D]\!]$.

Theorem 3.19 *Given an interpretation* $\mathcal{I}^\alpha \in \mathbb{I}_{\mathbb{A}}$ *and* $p(\overrightarrow{x}) \in \mathbb{PC}$,

$$\alpha^\tau(\mathcal{A}[\![A]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}) \le \mathcal{A}^\alpha[\![A]\!]_{\mathcal{I}^\alpha} \tag{3.5}$$

$$\alpha^\tau(\mathcal{D}[\![D]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}(p(\overrightarrow{x}))) \le \mathcal{D}^\alpha[\![D]\!]_{\mathcal{I}^\alpha}(p(\overrightarrow{x})) \tag{3.6}$$

Proof sketch. The complete proof of this result can be found in Appendix A. It proceeds by structural induction on the form of the agents by applying α^τ to the various cases of the concrete agent evaluation and, by correctness of α^τ, one obtains as over-approximation the corresponding cases of the definition of the abstract version. In particular, when the abstraction is applied to concrete constraints, the correctness is

guaranteed by our assumptions on the approximation relations $\rightarrowtail^+$ and $\rightarrowtail^-$. When the abstraction is applied to concrete auxiliary operators, the proof relies on their correctness, which is stated by means of four lemmas, also included in the appendix. □

3.3. From infinite to finite semantics

Since the domain of abstract conditional traces is not Noetherian (i.e., it admits infinite increasing chains), the computation of the abstract least fixpoint does not necessarily converge in finite time. Our solution is to use a *widening operator* [BHRZ05, CC77] that ensures convergence to an over-approximation of the abstract fixpoint in a finite number of steps.

In the following, we use a representation of sets of abstract conditional traces in terms of *conditional graphs*. These graphs are enriched with the information about the process calls, which is necessary to identify the part of the graph corresponding to each iteration of $\mathcal{D}^\alpha[\![D]\!]$ at the moment of applying the widening operator.

Definition 3.20 *A* conditional graph G *is a triple* $(Init, Nodes, Edges)$ *where*

- *Init is the set of initial nodes, each one labeled with a (unique) process symbol, denoted by* $init(G)$
- *Nodes is a set of nodes, each one containing a conditional step, and*
- *Edges is a set of edges between nodes that can be of two kinds: either simple edges* $n \to n'$*, or edges of the form* $n \overset{\rho}{\underset{p}{\Longrightarrow}} n'$ *representing a call to process* p *with variable renaming* ρ*. Edges represent the passage of one time unit.*

$\mathbb{G}$ *denotes the set of all conditional graphs. Moreover,* $n \not\to$ *denotes a node* n *that has no outgoing edges.*

We define the function $paths{:}\mathbb{G} \to \mathbb{A}$ which, given a conditional graph, returns the set of all paths of the graph. When an arc of the form $\overset{\rho}{\underset{p}{\Longrightarrow}}$ is traversed, a variant with fresh variables in the co-domain of the renaming ρ is applied to the nodes that follow in the path and the information of the store is propagated to the positive conditions, similarly to what happens when a process call is done. The order relation over graphs $\preccurlyeq$ is defined as $G_1 \preccurlyeq G_2 \iff paths(G_1) \leq paths(G_2)$. $(\mathbb{G}, \preccurlyeq, \curlyvee, \curlywedge, \top_{\mathbb{G}}, \bot_{\mathbb{G}})$ is a complete lattice where $\curlyvee$ is the least upper bound operator that joins a set of graphs by combining all the sequences that have a prefix in common in the same path, $\curlywedge$ is the greatest lower bound operator that returns the common parts of a set of graphs, $\bot_{\mathbb{G}}$ is the graph composed only of an empty initial node and $\top_{\mathbb{G}}$ is the graph such that $paths(\top_{\mathbb{G}}) = \mathbf{AT}$.

The semantics of a *tccp* process $p(\vec{x})$ can be seen as a conditional graph G with the initial node labeled with p and such that $paths(G) = \mathcal{F}^\alpha[\![D]\!](p(\vec{x}))$. The graph for the process $p(\vec{x})$ is built by linking the initial node of p to the nodes corresponding to the first conditional states of the semantics of an agent A such that $p(\vec{x}) :- A \in D$. The rest of the graph is built following the denotational semantics of Definition 3.18: each conditional state becomes a node in the graph and it is connected to the following one by a simple edge. When a call to a process $q(\vec{y})$ is found and the declaration $q(\vec{z}) :- A'$ is in D, an arrow $\overset{[\vec{z}/\vec{y}]}{\underset{q}{\Longrightarrow}}$ is added, thus linking the current node to the graph labeled with q by using the variable renaming $[\vec{z}/\vec{y}]$.

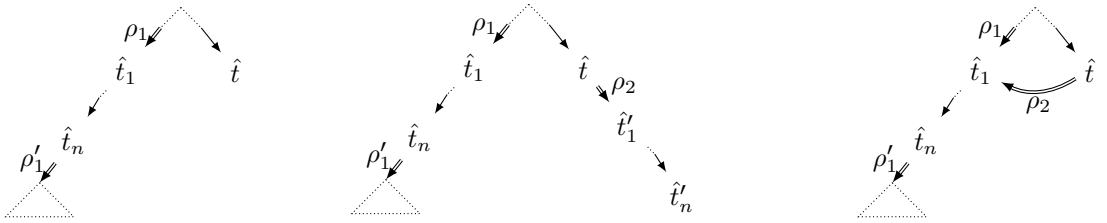
Now we are ready to define our widening operator. Widening operators provide an efficacious solution to the convergence problem by over-approximating infinite increasing chains in a finite number of steps. A *widening operator* [BHRZ05, CC77] on the lattice $(\mathbb{L}, \leq)$ is a partial function $\triangledown{:}\mathbb{L} \times \mathbb{L} \to \mathbb{L}$ satisfying:

(covering) for all $x, y \in \mathbb{L}$ such that $x \leq y$, $x \triangledown y$ is defined and $y \leq x \triangledown y$; and

(termination) for each increasing chain $x_0 \leq x_1 \leq \ldots$ the chain defined as $y_0 = x_0$ and $y_{i+1} = y_i \triangledown x_{i+1}$ is not strictly increasing.

We propose a widening operator[10] $\triangledown$ that looks for repeated patterns in consecutive iterations of $\mathcal{D}^\alpha[\![D]\!]$ and converges, in a finite number of steps, in a conditional graph that represents an over-approximation of

[10] In defining our widening operator, we follow the approach of [BHRZ05] instead of the original in [CC77].



(a) G_1 (b) G_2 (c) $G_1 \triangledown G_2$

Figure 3. The graph widening behavior.

the abstract fixpoint $\mathcal{F}^\alpha$. In the sequel, we abuse in notation and write $\hat{t} \underset{p}{\overset{\rho_2}{\Longrightarrow}} \hat{t}'_1 \to \cdots \to \hat{t}'_n$ to denote the set of the edges occurring in this path, i.e., $\{\hat{t} \underset{p}{\overset{\rho_2}{\Longrightarrow}} \hat{t}'_1, \hat{t}'_1 \to \hat{t}'_2 \ldots, \hat{t}'_{n-1} \to \hat{t}'_n\}$.

Definition 3.21 (Graph widening) *Let $G_1, G_2 \in \mathbb{G}$ such that $G_1 \preccurlyeq G_2$. The graph widening of G_1 w.r.t. G_2 is defined as $G_1 \triangledown G_2 \coloneqq G_1 \vee (I, N, E)$ where $I \coloneqq init(G_2)$, N is the set of nodes that occur in the set of edges E, and*

$$
\begin{aligned}
E \coloneqq \Big\{ \hat{t} \underset{p}{\overset{\rho_2}{\Longrightarrow}} \hat{t}_1 \mid & \text{ it exists a subpath in } G_2 \text{ of the form } \hat{t} \underset{p}{\overset{\rho_2}{\Longrightarrow}} \hat{t}'_1 \ldots \hat{t}'_n \not\to \text{ s.t. an edge } \underset{p}{\Longrightarrow} \text{ does not occur in} \\
& \hat{t}'_1 \ldots \hat{t}'_n \text{ and it exists a subpath in } G_1 \text{ of the form } \underset{p}{\overset{\rho_1}{\Longrightarrow}} \hat{t}_1 \ldots \hat{t}_n \underset{p}{\overset{\rho'_1}{\Longrightarrow}}, \text{ s.t. an edge } \underset{p}{\Longrightarrow} \text{ does} \\
& \text{not occur in } \hat{t}_1 \ldots \hat{t}_n \text{ and } \forall 1 \le i \le n \ \rho_1(\hat{t}_i) = \rho_2(\hat{t}'_i) \Big\} \cup \\
\bigcup \Big\{ \hat{t} \underset{p}{\overset{\rho_2}{\Longrightarrow}} \hat{t}'_1 \to \cdots \to \hat{t}'_n \mid & \text{ it exists a subpath in } G_2 \text{ on the form } \hat{t} \underset{p}{\overset{\rho_2}{\Longrightarrow}} \hat{t}'_1 \ldots \hat{t}'_n \not\to \text{ s.t. in } \hat{t}'_1 \ldots \hat{t}'_n \text{ it does} \\
& \text{not occur an edge } \underset{p}{\Longrightarrow} \text{ and it does not exist a subpath in } G_1 \text{ of the form } \underset{p}{\overset{\rho_1}{\Longrightarrow}} \hat{t}_1 \ldots \hat{t}_n \underset{p}{\overset{\rho'_1}{\Longrightarrow}}, \text{ s.t.} \\
& \text{in } \hat{t}_1 \ldots \hat{t}_n \text{ it does not occur an edge } \underset{p}{\Longrightarrow} \text{ and } \forall 1 \le i \le n \ \rho_1(\hat{t}_i) = \rho_2(\hat{t}'_i) \Big\}
\end{aligned}
$$

At each iteration, the widening checks if a suffix r of a path b in the graph of a process p (which corresponds to the trace produced at the last iteration of p) has already appeared in a previous iteration of p (modulo variables renaming). In this case, it adds an edge, labeled with the necessary variable renaming ρ_2, from the node $\hat{t}$ precedent to the pattern r to the first node of the equivalent pattern found in the previous widening iteration (first case of Definition 3.21). Otherwise, if no equivalent pattern (modulo variable renaming) is found, the path b is added to the graph (second case of Definition 3.21). Figure 3 shows a graphical representation of the graph widening behavior. To improve readability, in the figure we assume that all process calls involve the same process, thus we just include the renaming for variables in the edges.

Lemma 3.22 *If the underlying abstract Cylindric Constraint System is Noetherian, then the operator $\triangledown$ is a widening operator on $\mathbb{G}$.*

Proof sketch. The covering property is a consequence of the fact that the branches of G_2 that are not included by the widening are already present in G_1 modulo variable renaming; that is the reason why a direct edge is added from the last node before the repetition to the equivalent branch detected in G_1.

Termination of the widening is a consequence of the properties of the abstract constraint systems and of the finiteness of the program syntax. By definition, just a finite number of conditional steps can be computed, thus iteration's length is finite. Furthermore, when a repeated pattern is detected, that (possibly cyclic) branch is not further expandable. □

Because of Lemma 3.22 and the results in [BHRZ05] is guaranteed that, for any *tccp* set of declarations D, the chain

$$
I_0 = \{\epsilon\} \qquad I_{i+1} = \begin{cases} I_i & \text{if } \mathcal{D}^\alpha[\![D]\!]_{I_i} \le I_i \\ I_i \triangledown (I_i \vee \mathcal{D}^\alpha[\![D]\!]_{I_i}) & \text{otherwise} \end{cases}
$$

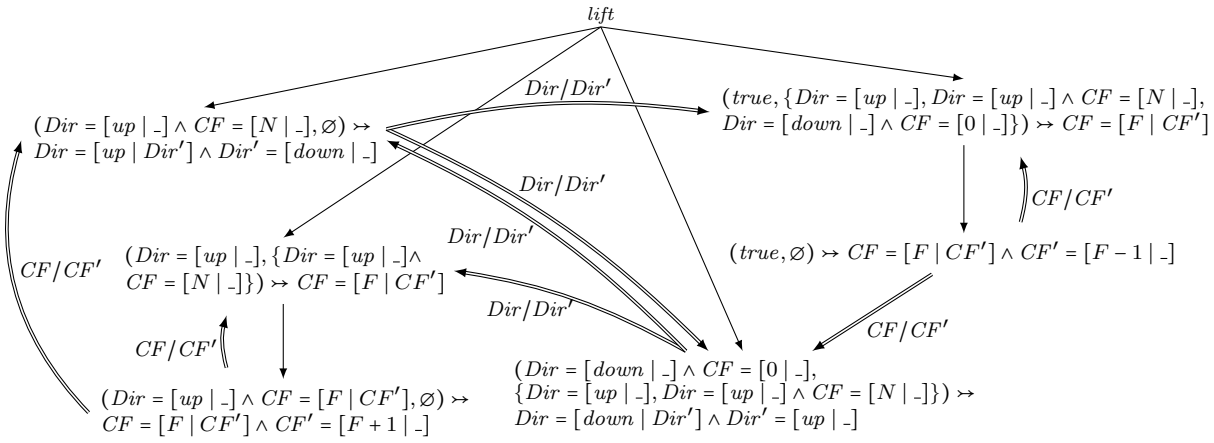


Figure 4. Graph representation of the abstract semantics of the *lift* process.

converges to a graph which is a correct approximation of the abstract semantics in a finite number of steps. That graph contains an initial node for each process declaration such that the subgraph reachable from the initial node represents the behaviors of the corresponding process. Subgraphs corresponding to different processes are linked by edges with renamings when process calls occur.

Example 3.23 Figure 4 shows the conditional graph corresponding to the abstract semantics of the *lift* process. We abstract streams of the concrete Constraint System by posing a depth limit for streams, i.e., we keep the first k values of a stream, and then we use the top of the domain. All other constraints are abstracted to themselves. The resulting abstract Constraint System is thus finite.

Due to the application of the widening operator it can be noted how the recursive calls (represented as triangles in Figure 1) are replaced in Figure 4 with the (set of) arcs pointing to the possible continuations of the computation. ■

4. Abstract analysis with an over-approximation

The abstract semantics we have proposed so far is an over-approximation of the concrete semantics. Thus, it allows us to check *universal properties*, i.e., properties that all the possible behaviors of the system must satisfy. For instance, it is possible to analyze some temporal properties such as safety (i.e., something bad never happens) or liveness (i.e., something good eventually happens) or to check if a program never suspends. In order to check whether some invariant property is satisfied by our program, it is necessary to check if every node of the graph respects this property. The properties that can be checked strongly depend on the abstraction of the constraint system. If we want to guarantee that a given abstract constraint $\hat{c}$ never holds in a computation, we need to check that for every node, either its store is in contradiction with $\hat{c}$, or its negative condition contains a store that satisfies $\hat{c}$ or the positive condition $\hat{\eta}$ is in contradiction with $\hat{c}$ (i.e., $\hat{\eta} \hat{\otimes} \hat{c} = \hat{false}$). This ensures that, for every possible input, $\hat{c}$ is never produced in the computation.

Similarly, in order to check if an abstract constraint $\hat{c}$ is always entailed by the current store, it is sufficient to check if for each conditional step occurring in the graph of the form $(\hat{\eta}, \check{\eta}) \rightarrowtail \hat{d}$, the positive condition merged with the store entails $\hat{c}$ (i.e., $\hat{\eta} \hat{\otimes} \hat{d} \hat{\vdash} \hat{c}$). This ensures that for every possible initial constraint, $\hat{c}$ is entailed by the store.

Example 4.1 We may be interested in proving several invariant properties on the *lift* process in Example 2.1. For instance, we can try to verify that "the current floor stream *CF* never gets a negative number". To this end, we check all the conditions in the graph in Figure 4, and since we find (at least) a node that does not contradict that *CF* is negative (see the first node of the right branch), we conclude that it cannot be ensured that the *lift* process respects this safety property. As a matter of fact, provided we start the

computation with an initial state where CF is initialized to a negative number, then the last else branch of the program can be taken, and CF would remain negative in the subsequent trace.

Consider now the invariant property "each time the direction of the lift is updated, also its floor is updated". In this case, it can be noticed that all the conditional steps in Figure 4 satisfy this property, since whenever the positive condition in the step is merged with the store, it entails that Dir has a value, then it is also entailed that CF is instantiated. ■

Verifying liveness properties is harder since it involves analyzing unknown length sequences of steps. For instance, given a process $p(\overrightarrow{x})$, assume that we want to check that "every time an abstract constraint $\hat{c}$ holds, then it exists a future state where another abstract constraint $\hat{d}$ holds". Given the conditional graph for $p(\overrightarrow{x})$, this property would hold if for each node labeled with a conditional step whose positive condition and store entails $\hat{c}$ then all paths starting from such node contain a conditional step whose positive condition and store entails $\hat{d}$.

Example 4.2 Observe that *lift* process in Example 2.1 satisfies the property "every time the current floor is 0 and the direction is *down*, the direction will be *up* eventually". In fact, the first node of the third branch from the left in Figure 4 is the sole step that contains in its positive condition $CF = [0 \mid \_]$ and $Dir = [down\_]$. Furthermore, for each possible path from this node we find a conditional step where $Dir = [up \mid \_]$ appears in the positive condition or in the store.

Another interesting liveness property that can be analyzed on the *lift* process is "whenever the current floor is 0 it exists a future state when this value changes", i.e., we do not stay indefinitely in floor 0. ■

Since the number of nodes in the graph is finite, the aforementioned analysis terminates in a finite number of steps.

Let us now analyze non-suspension. Non-suspension analysis consists in ensuring that no execution of a *tccp* program suspends. In conditional graphs, in order to check whether $p(\overrightarrow{x})$ never suspends, it is sufficient to check that there is no node N in G labeled with a *stutt* construct with an outgoing arc pointing to N itself. Inversely, if the graph contains a stuttering node, we can not guarantee suspension, due to over-approximation of the semantics.

Example 4.3 Consider the semantics of the *lift* process in Figure 4. It is worth noting that the graph does not contain any node labeled with *stutt*. Therefore, we can ensure that the *lift* process never suspends. ■

In the previous paragraphs, we have discussed the analysis of the lift process in isolation, without taking into account the rest of processes which are concurrently in execution. The verification of properties on systems composed of more than one process involves the construction of complex graphs in which each node contains the positive and negative conditions along with the accumulated store obtained by the synchronous execution of all active processes. Clearly, the size of this graph is a key parameter to determine the complexity of verification algorithms based on graph exploration. For example, assume that we try to analyze properties on the system composed by processes *lift* and *pssngr* of our guiding example. The graph corresponding to the initial process

$$main(N, O, D) :- \exists\, CF, Dir, St \left(lift(N, CF, Dir) \parallel pssngr(CF, O, D, St) \parallel \right.$$
$$\left. \mathsf{tell}(CF = [0 \mid \_]) \parallel \mathsf{tell}(Dir = [up \mid \_]) \parallel \mathsf{tell}(St = [wait|\_]) \right)$$

contains nodes where both the *lift* and the *pssngr* evolve synchronously following the behavior defined by the semantics in Section 3. The size of this graph strongly depends on the initial value of variable N, i.e., the number of the floors, since the *lift* process iteratively changes its current floor in the range $[0..N]$ and, each time it changes, the *pssngr* process compares the current floor with the parameters origin and destination. In order to model all the possibilities, the parallel composition results in a graph that can be seen as the composition of the *lift* graph in Figure 4 (fed with the initial constraints given by the two tell agents, and unfolded by making the possible floors explicit) with the *pssngr* graph. We could also complicate the structure of the system by considering more than one lift moving up and down in the building or adding more passengers. In any case, the analysis of real systems would clearly benefit from data abstraction following the methodology described in Section 3.1.

Example 4.4 As an example of data abstraction for the guiding example, consider the Floors abstract domain of Example 3.2. With this abstraction, we could construct the abstract graph of the *main* agent

above which is independent of the actual number N of building floors. Clearly, the price to pay is the loss of precision when analyzing some properties on the system. For instance, we could prove that the system *lift*/*pssngr* does not block irrespective of the number of floors in the building. It could be also possible to analyze properties which are not affected by the abstraction such as "if the *lift* is at floor 0 and the *lift* direction is *down*, then the *lift* does not move until the direction changes to *up*."

However, observe that due to the imprecision when carrying out operations on abstract values, the abstract graph contains spurious behaviors that make impossible to directly prove some liveness properties. For instance, since adding 1 to the abstract value $notLower_x$ returns $notLower_x$, the abstract graph includes paths where the current floor is always increasing, which is clearly unrealistic. Several techniques may be utilized to eliminate these false behaviors. For instance, in [CGJ+00], the semantics is gradually refined by using spurious counterexamples. It is also possible to prune the part of the graph explored by assuming fairness conditions as described in [GMP02]. For example, by imposing condition "the *lift* will reach floor 0 infinitely often", it could be possible to prove liveness properties as "if the origin and destination floors for the *pssngr* are 0 and N, respectively, then the *pssngr* will enter in the *lift* at floor 0 at some future time, and she will leave the *lift* at floor N, eventually". ■

5. Related work

The recent work [CGTV15] was the first attempt to propose a program analysis framework based on abstract interpretation for a concurrent constraint language adhering to the characteristics of *tccp* (negative information, non-determinism and infinite behaviors). The new proposed framework improves the applicative domain of the work in [CGTV15]. More specifically, we have relaxed the condition on the abstract domain so that now we can use abstract domains that better capture the relations of practical interest in the concrete domain. For instance, the domain of Example 3.2 does not satisfy the conditions required in [CGTV15].

Previously, in [FOP15], a framework for dataflow analysis of programs written in two other languages of the *ccp* family, *tcc* and *utcc*, is presented. The two main differences between these two languages and *tccp* are the notion of time (*tcc* and *utcc* use dedicated timing constructs) and determinism (*vs.* non-determinism of *tccp*). Moreover, in the case studies, [FOP15] uses a $depth(k)$ abstraction to ensure convergence, which consists in a non-selective cut at some point in time (instead of the selective cut that we can use by widening like in Example 3.23).

In [FV06], it was defined a model checking algorithm for *tccp* which allowed us to verify timed-depending properties. Their algorithm was based on the exploration of a graph representation of the program behavior which resembles the graph representation of the semantics defined in this paper. Thus we could as well employ our graph representation to perform (an efficient) model checking. Note however that the abstract semantics that we propose now is not limited to the verification of temporal properties.

Finally, [AGPV05] proposes an abstract semantic framework for *tccp* that, differently from our approach, was based on source-to-source transformations. The two approaches are completely different: [AGPV05] aimed at using the concrete semantics to execute the transformed (abstract) program. This could be done thanks to a non-trivial transformation of the program (an analysis on the structure of the program was necessary as a preprocess of the transformation). Our approach aims at defining an abstract semantics that, thanks to the characteristics of the concrete denotational semantics, is guaranteed to be correct and we argue that is precise enough to allow the definition of interesting analyses.

6. Conclusions and future work

We have proposed a program analysis framework based on an abstract semantics that, together with a widening operator, is suitable for the definition of different analyses for *full tccp* programs. This is a difficult task because of the presence of infinite computations, use of negative information and non-determinism. However, it is essential to consider these features of the language since these are the ones that make *tccp* well-suited to model reactive systems.

The abstract semantics is an over-approximation, which makes possible to define analysis tools for universal properties. To the best of our knowledge, this is the first proposal that defines an analysis which adaptively ensures termination depending on the program (by means of widening). This should give better results than the non-selective approaches.

We have also improved the framework previously defined in [CGTV15] by relaxing the properties of the abstract domain and we have shown its applicability by means of examples.

This work culminates the first step towards our final goal of defining a rich abstract semantic framework for the analysis of *tccp* programs. As future work, we are interested in defining an under-approximating framework for *tccp*. Under-approximations do not capture all possible program's behaviors, but no spurious ones are included. These kind of abstractions allows one to analyze *existential* properties, for instance the existence of a suspension trace.

References

[AGPV05] M. Alpuente, M.M. Gallardo, E. Pimentel, and A. Villanueva. A Semantic Framework for the Abstract Model Checking of tccp Programs. *Theoretical Computer Science*, 346(1):58–95, 2005.

[BHRZ05] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise Widening Operators for Convex Polyhedra. *Science of Computer Programming*, 58(1-2):28–56, 2005.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, Los Angeles, California, January 17–19*, pages 238–252, New York, NY, USA, 1977. ACM Press.

[CGJ+00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, number 1855 in Lecture Notes in Computer Science, pages 154–169. Springer Verlag, 2000.

[CGTV15] M. Comini, M.M. Gallardo, L. Titolo, and A. Villanueva. Abstract Analysis of Universal Properties for tccp. In M. Falaschi, editor, *Logic-based Program Synthesis and Transformation, 25th International Symposium, LOPSTR 2015. Revised Selected Papers*, volume 9527 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 2015.

[CTV11] M. Comini, L. Titolo, and A. Villanueva. Abstract Diagnosis for Timed Concurrent Constraint programs. *Theory and Practice of Logic Programming*, 11(4-5):487–502, 2011.

[CTV13] M. Comini, L. Titolo, and A. Villanueva. A Condensed Goal-Independent Bottom-Up Fixpoint Modeling the Behavior of tccp. Technical report, DSIC, Universitat Politècnica de València. Available at http://riunet.upv.es/handle/10251/34328, 2013.

[dBGM00] F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Timed Concurrent Constraint Language. *Information and Computation*, 161(1):45–83, 2000.

[FGMP93] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Compositional Analysis for Concurrent Constraint Programming. In *Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 210–221, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[FOP15] M. Falaschi, C. Olarte, and C. Palamidessi. Abstract Interpretation of Temporal Concurrent Constraint Programs. *Theory and Practice of Logic Programming (TPLP)*, 15(3):312–357, 2015.

[FV06] M. Falaschi and A. Villanueva. Automatic verification of timed concurrent constraint programs. *Theory and Practice of Logic Programming*, 6(3):265–300, 2006.

[GMP02] M. M. Gallardo, P. Merino, and E. Pimentel. Refinement of LTL formulas for abstract model checking. In *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings*, pages 395–410, 2002.

[Sar93] V. A. Saraswat. *Concurrent Constraint Programming*. The MIT Press, Cambridge, Mass., 1993.

[SRP91] V. A. Saraswat, M. Rinard, and P. Panangaden. The Semantic Foundations of Concurrent Constraint Programming. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–352, New York, NY, USA, 1991. ACM.

[ZGL97] E. Zaffanella, R. Giacobazzi, and G. Levi. Abstracting Synchronization in Concurrent Constraint Programming. *Journal of Functional and Logic Programming*, 1997(6), 1997.

A. Proofs

The two abstract propagation operators are correct w.r.t. the concrete ones, as formally stated by the following lemma.

Lemma A.1 (Correctness of strong and weak abstract propagation) *Let* $r \in \mathbf{CT}$ *and* $c \in \mathbf{C}$. *Then* $\alpha^\tau(r{\downarrow_c}) \leq \alpha^\tau(r)\hat{\downarrow}_{\tau(c)}$ *and* $\alpha^\tau(r{\downarrow_c}) \leq \alpha^\tau(r)\hat{\downarrow}_{\tau(c)}$.

Proof. First we prove that $\alpha^\tau(r{\downarrow_c}) \leq \alpha^\tau(r)\hat{\downarrow}_{\tau(c)}$. Note that the concrete and abstract versions of the operators are structurally identical. They differ in using and abstract constraints and the related abstract operations instead of their concrete versions. The proof proceeds by structural induction, proving that, for each case, the abstract conditional traces resulting by applying the abstract operators is greater or equal (less precise) than the traces resulting from applying α^τ to the output of the concrete operation.

Let us first recall the definition for the concrete operator from [CTV13].

Definition A.2 (Concrete propagation operator) *Let $r \in \mathbf{CT}$ and $c \in \mathbf{C}$. We define the* propagation *of c in r, written $r{\downarrow_c}$, by structural induction as $\epsilon{\downarrow_c} = \epsilon$, $\boxtimes{\downarrow_c} = \boxtimes$, and*

$$((\eta^+, \eta^-) \rightarrowtail d \cdot r'){\downarrow_c} = \begin{cases} (c \otimes \eta^+, \eta^-) \rightarrowtail d \otimes c \cdot (r'{\downarrow_c}) & \text{if } c \gg (\eta^+, \eta^-),\ c \otimes d \neq \mathit{false} \\ (c \otimes \eta^+, \eta^-) \rightarrowtail \mathit{false} \cdot \boxtimes & \text{if } c \gg (\eta^+, \eta^-),\ c \otimes d = \mathit{false} \end{cases}$$

$$(\mathit{stutt}(\eta^-) \cdot r'){\downarrow_c} = \mathit{stutt}(\eta^-) \cdot (r'{\downarrow_c}) \quad \text{if } \forall c^- \in \eta^-.\, c \not\vdash c^-$$

Now we proceed by cases:

$r = \epsilon$ $\forall c \in \mathbf{C}.\ \alpha^\tau(\epsilon{\downarrow_c}) = \epsilon = \alpha^\tau(\epsilon)\hat{\downarrow}_{\tau(c)}$.

$r = \boxtimes$ $\forall c \in \mathbf{C}.\ \alpha^\tau(\boxtimes{\downarrow_c}) = \boxtimes = \alpha^\tau(\boxtimes)\hat{\downarrow}_{\tau(c)}$.

$r = (\eta^+, \eta^-) \rightarrowtail d \cdot r'$ In case $c \not\gg (\eta^+, \eta^-)$, $r{\downarrow_c}$ and $\alpha^\tau(r{\downarrow_c})$ are not defined. Otherwise, if $c \gg (\eta^+, \eta^-)$ we can distinguish two cases.

$c \otimes d \neq \mathit{false}$ $\alpha^\tau(r) = (\tau(\eta^+), \bar{\tau}(\eta^-)) \rightarrowtail \tau(d) \cdot \alpha^\tau(r')$. Moreover, from Equation (3.2), $c \gg (\eta^+, \eta^-) \Rightarrow \tau(c) \hat{\gg} (\tau(\eta^+), \bar{\tau}(\eta^-))$ and from the insertion between concrete and abstract domain, $c \otimes d \neq \mathit{false} \Rightarrow \tau(c) \hat{\otimes} \tau(d) \neq \hat{\mathit{false}}$.

$$\begin{aligned}
\alpha^\tau(r{\downarrow_c}) &= \alpha^\tau(((\eta^+, \eta^-) \rightarrowtail d \cdot r'){\downarrow_c}) \\
&\quad [\text{since } c \gg (\eta^+, \eta^-) \text{ and } c \otimes d \neq \mathit{false}] \\
&= \alpha^\tau((c \otimes \eta^+, \eta^-) \rightarrowtail c \otimes d \cdot (r'{\downarrow_c})) \\
&\quad [\text{by Definition 3.8 } (\alpha^\tau)] \\
&= (\tau(c \otimes \eta^+), \bar{\tau}(\eta^-)) \rightarrowtail \tau(c \otimes d) \cdot \alpha^\tau(r'{\downarrow_c}) \\
&\quad [\text{by Inductive Hypothesis}] \\
&\leq (\tau(c \otimes \eta^+), \bar{\tau}(\eta^-)) \rightarrowtail \tau(c \otimes d) \cdot \alpha^\tau(r')\hat{\downarrow}_{\tau(c)} \\
&\quad [\text{by the abstract domain properties and order}] \\
&\leq (\tau(c) \hat{\otimes} \tau(\eta^+), \{\tau(\eta_n^-) \mid \eta_n^- \in \eta^-\}) \rightarrowtail (\tau(c) \hat{\otimes} \tau(d)) \cdot \alpha^\tau(r')\hat{\downarrow}_{\tau(c)} \\
&\quad [\text{since } \tau(c) \hat{\gg} (\tau(\eta^+), \bar{\tau}(\eta^-)) \text{ and } \tau(c) \hat{\otimes} \tau(d) \neq \hat{\mathit{false}}, \text{ and by Definition 3.9}] \\
&= \alpha^\tau(r)\hat{\downarrow}_{\tau(c)}
\end{aligned}$$

$c \otimes d = \mathit{false}$ Similarly to the previous case, by Equation (3.2), $c \gg (\eta^+, \eta^-) \Rightarrow \tau(c) \hat{\gg} (\tau(\eta^+), \bar{\tau}(\eta^-))$ and from the insertion between concrete and abstract domain, $c \otimes d = \mathit{false} \Rightarrow \tau(c) \hat{\otimes} \tau(d) = \hat{\mathit{false}}$.

$$\begin{aligned}
\alpha^\tau(r{\downarrow_c}) &= \alpha^\tau(((\eta^+, \eta^-) \rightarrowtail d \cdot r'){\downarrow_c}) \\
&\quad [\text{since } c \gg (\eta^+, \eta^-) \text{ and } c \otimes d = \mathit{false}] \\
&= \alpha^\tau((c \otimes \eta^+, \eta^-) \rightarrowtail \mathit{false} \cdot \boxtimes) \\
&\quad [\text{by Definition 3.8 } (\alpha^\tau)] \\
&= (\tau(c \otimes \eta^+), \bar{\tau}(\eta^-)) \rightarrowtail \hat{\mathit{false}} \\
&\quad [\text{by the abstract domain properties}] \\
&\leq (\tau(c) \hat{\otimes} \tau(\eta^+), \{\tau(\eta_n^-) \mid \eta_n^- \in \eta^-\}) \rightarrowtail \hat{\mathit{false}} \\
&\quad [\text{since } \tau(c) \hat{\gg} (\tau(\eta^+), \bar{\tau}(\eta^-)) \text{ and } \tau(c) \hat{\otimes} \tau(d) = \hat{\mathit{false}}, \text{ and by Definition 3.9}] \\
&= \alpha^\tau(r)\hat{\downarrow}_{\tau(c)}
\end{aligned}$$

$r = \mathit{stutt}(\eta^-) \cdot r'$ In case $c \not\gg (\eta^+, \eta^-)$, $r{\downarrow_c}$ and $\alpha^\tau(r{\downarrow_c})$ are not defined. Otherwise, if $c \gg (\eta^+, \eta^-)$, we have:

$$\begin{aligned}
\alpha^\tau(r{\downarrow_c}) &= \alpha^\tau(\mathit{stutt}(\eta^-) \cdot r'{\downarrow_c}) \\
&\quad [\text{by Definition 3.8 } (\alpha^\tau)]
\end{aligned}$$

$$
\begin{aligned}
&= \mathit{stutt}(\overline{\tau}(\eta^-)) \cdot \alpha^\tau(r'{\downarrow_c}) \\
&\qquad [\text{by Inductive Hypothesis}] \\
&\leq \mathit{stutt}(\overline{\tau}(\eta^-)) \cdot \alpha^\tau(r')\hat{\downarrow}_{\tau(c)} \\
&= \mathit{stutt}(\{\tau(\eta_n^-) \mid \eta_n^- \in \eta^-\}) \cdot \alpha^\tau(r')\hat{\downarrow}_{\tau(c)} \\
&\qquad [\text{by Definition 3.9}] \\
&= \alpha^\tau(r)\hat{\downarrow}_{\tau(c)}
\end{aligned}
$$

Now we prove that $\alpha^\tau(r{\downarrow_c}) \leq \alpha^\tau(r)\hat{\downarrow}_{\tau(c)}$ by induction on the structure of r. Again, we first recall the definition for the concrete operator from [CTV13].

Definition A.3 (Concrete weak propagation operator) *Let $r \in \mathbf{CT}$ and $c \in \mathbf{C}$. We define the* weak propagation *of c in r, denoted $r{\downarrow_c}$, as $\epsilon{\downarrow_c} := \epsilon$, $\boxtimes{\downarrow_c} := \boxtimes$, and*

$$
\begin{aligned}
&((\eta^+, \eta^-) \rightarrowtail d \cdot r'){\downarrow_c} := (c \otimes \eta^+, \eta^-) \rightarrowtail d \cdot (r'{\downarrow_c}) && \text{if } c \gg (\eta^+, \eta^-) \\
&(\mathit{stutt}(\eta^-) \cdot r'){\downarrow_c} := \mathit{stutt}(\eta^-) \cdot (r'{\downarrow_c}) && \text{if } \forall c^- \in \eta^-.\, c \not\vdash c^-
\end{aligned}
$$

Now we proceed by cases:

$\underline{\boldsymbol{r = \epsilon}}$ $\forall c \in \mathbf{C}.\ \alpha^\tau(\epsilon{\downarrow_c}) = \epsilon = \alpha^\tau(\epsilon)\hat{\downarrow}_{\tau(c)}$.

$\underline{\boldsymbol{r = \boxtimes}}$ $\forall c \in \mathbf{C}.\ \alpha^\tau(\boxtimes{\downarrow_c}) = \boxtimes = \alpha^\tau(\boxtimes)\hat{\downarrow}_{\tau(c)}$.

$\underline{\boldsymbol{r = (\eta^+, \eta^-) \rightarrowtail d \cdot r'}}$ In case $c \not\gg (\eta^+, \eta^-)$, $r{\downarrow_c}$ and $\alpha^\tau(r{\downarrow_c})$ are not defined. Otherwise, from Equation (3.2), $c \gg (\eta^+, \eta^-) \Rightarrow \tau(c) \hat{\gg} (\tau(\eta^+), \overline{\tau}(\eta^-))$.

$$
\begin{aligned}
\alpha^\tau(r{\downarrow_c}) &= \alpha^\tau(((\eta^+, \eta^-) \rightarrowtail d \cdot r'){\downarrow_c}) \\
&\qquad [\text{since } c \gg (\eta^+, \eta^-)] \\
&= \alpha^\tau((c \otimes \eta^+, \eta^-) \rightarrowtail d \cdot (r'{\downarrow_c})) \\
&\qquad [\text{by Definition 3.8 } (\alpha^\tau)] \\
&= (\tau(c \otimes \eta^+), \overline{\tau}(\eta^-)) \rightarrowtail \tau(d) \cdot \alpha^\tau(r'{\downarrow_c}) \\
&\qquad [\text{by Inductive Hypothesis}] \\
&\leq (\tau(c \otimes \eta^+), \overline{\tau}(\eta^-)) \rightarrowtail \tau(d) \cdot \alpha^\tau(r')\hat{\downarrow}_{\tau(c)} \\
&\qquad [\text{by the abstract domain properties}] \\
&\leq (\tau(c) \hat{\otimes} \tau(\eta^+), \{\tau(\eta_n^-) \mid \eta_n^- \in \eta^-\}) \rightarrowtail \tau(d) \cdot \alpha^\tau(r')\hat{\downarrow}_{\tau(c)} \\
&\qquad [\text{since } \tau(c) \hat{\gg} (\tau(\eta^+), \overline{\tau}(\eta^-)), \text{ and by Definition 3.11}] \\
&= \alpha^\tau(r)\hat{\downarrow}_{\tau(c)}
\end{aligned}
$$

$\underline{\boldsymbol{r = \mathit{stutt}(\eta^-) \cdot r'}}$ In case $c \not\gg (\eta^+, \eta^-)$, $r{\downarrow_c}$ and $\alpha^\tau(r{\downarrow_c})$ are not defined. Otherwise, if $c \gg (\eta^+, \eta^-)$, we have:

$$
\begin{aligned}
\alpha^\tau(r{\downarrow_c}) &= \alpha^\tau((\mathit{stutt}(\eta^-) \cdot r'){\downarrow_c}) \\
&\qquad [\text{since } c \gg (\eta^+, \eta^-)] \\
&= \alpha^\tau(\mathit{stutt}(\eta^-) \cdot r'{\downarrow_c}) \\
&\qquad [\text{by Definition 3.8 } (\alpha^\tau)] \\
&= \mathit{stutt}(\overline{\tau}(\eta^-)) \cdot \alpha^\tau(r'{\downarrow_c}) \\
&\qquad [\text{by Inductive Hypothesis}] \\
&\leq \mathit{stutt}(\{\tau(\eta_n^-) \mid \eta_n^- \in \eta^-\}) \cdot \alpha^\tau(r')\hat{\downarrow}_{\tau(c)} \\
&\qquad [\text{by Definition 3.11}] \\
&= \alpha^\tau(r)\hat{\downarrow}_{\tau(c)}
\end{aligned}
$$

□

The following lemma states the soundness of $\hat{\|}$ w.r.t. the concrete parallel composition operator.

Lemma A.4 (Correctness of the abstract parallel) *Let* $r_1, r_2 \in \mathbb{C}$. *Then* $\alpha^\tau(r_1 \mathbin{\bar{\|}} r_2) \leq \alpha^\tau(r_1) \mathbin{\hat{\|}} \alpha^\tau(r_2)$

Proof. First let us recall the definition for the concrete operator from [CTV13].

Definition A.5 (Concrete parallel composition) *The* parallel composition *partial operator* $\bar{\|}: \mathbf{CT} \times \mathbf{CT} \to \mathbf{CT}$ *is the commutative closure of the following partial operation defined by structural induction as*[11] $r \mathbin{\bar{\|}} \epsilon := r$, $r \mathbin{\bar{\|}} \boxtimes := r$ *and*

$$(stutt(\eta_1^-) \cdot r_1') \mathbin{\bar{\|}} (stutt(\eta_2^-) \cdot r_2') := stutt(\eta_1^- \uplus \eta_2^-) \cdot (r_1' \mathbin{\bar{\|}} r_2')$$

Moreover, if $\eta_1 \otimes \eta_2$ *is consistent, then*

$$(\eta_1 \rightarrowtail c_1 \cdot r_1') \mathbin{\bar{\|}} (\eta_2 \rightarrowtail c_2 \cdot r_2') := \begin{cases} \eta_1 \otimes \eta_2 \rightarrowtail c_1 \otimes c_2 \cdot ((r_1'{\downarrow_{\eta_2^+}}{\downarrow_{c_2}}) \mathbin{\bar{\|}} (r_2'{\downarrow_{\eta_1^+}}{\downarrow_{c_1}})) & \text{if } c_1 \otimes c_2 \neq false \\ \eta_1 \otimes \eta_2 \rightarrowtail false \cdot \boxtimes & \text{if } c_1 \otimes c_2 = false, \end{cases}$$

Finally, if $\forall c^- \in \eta_2^- . \eta_1^+ \nvdash c^-$, *then*

$$((\eta_1^+, \eta_1^-) \rightarrowtail c_1 \cdot r_1') \mathbin{\bar{\|}} (stutt(\eta_2^-) \cdot r_2') := (\eta_1^+, \eta_1^- \mathbin{\hat{\uplus}} \eta_2^-) \rightarrowtail c_1 \cdot (r_1' \mathbin{\bar{\|}} (r_2'{\downarrow_{\eta_1^+}}{\downarrow_{c_1}}))$$

Note that the concrete and abstract versions of this operator are structurally identical. The abstract version is obtained from the concrete by replacing concrete constraints and operations on them by their abstract versions.

The proof proceeds by structural induction on the structure of r_1. The cases for r_2 are symmetric.

$r_1 = \epsilon$ and any r_2 $\alpha^\tau(r_1 \mathbin{\bar{\|}} r_2) = \alpha^\tau(r_2) = \epsilon \mathbin{\hat{\|}} \alpha^\tau(r_2) = \alpha^\tau(r_1) \mathbin{\hat{\|}} \alpha^\tau(r_2)$.

$r_1 = \boxtimes$ and any r_2 $\alpha^\tau(r_1 \mathbin{\bar{\|}} r_2) = \alpha^\tau(r_2) = \boxtimes \mathbin{\hat{\|}} \alpha^\tau(r_2) = \alpha^\tau(r_1) \mathbin{\hat{\|}} \alpha^\tau(r_2)$.

$r_1 = \eta_1 \rightarrowtail c_1 \cdot r_1'$ and $r_2 = \eta_2 \rightarrowtail c_2 \cdot r_2'$ In case $\eta_1 \otimes \eta_2$ is not a consistent condition $r_1 \mathbin{\bar{\|}} r_2$ is not defined and, as a consequence, also $\alpha^\tau(r_1 \mathbin{\bar{\|}} r_2)$ is not defined. Otherwise, if $\eta_1 \otimes \eta_2$ is a consistent condition, by the properties of the abstract domain and Equation 3.2, its abstraction is consistent too, thus, we can distinguish two cases.

$c_1 \otimes c_2 \neq false$ From the insertion between the concrete and abstract domains, it follows that $\tau(c_1) \hat{\otimes} \tau(c_2) \neq \hat{false}$.

$$\begin{aligned}
&\alpha^\tau(r_1 \mathbin{\bar{\|}} r_2) = \\
&= \alpha^\tau((\eta_1 \otimes \eta_2) \rightarrowtail c_1 \otimes c_2 \cdot (r_1'{\downarrow_{\eta_2^+}}{\downarrow_{c_2}} \mathbin{\bar{\|}} r_2'{\downarrow_{\eta_1^+}}{\downarrow_{c_1}})) \\
&\quad [\text{by Definition 3.8 } (\alpha^\tau)] \\
&= (\tau(\eta_1^+ \otimes \eta_2^+), \bar{\tau}(\eta_1^- \uplus \eta_2^-)) \rightarrowtail \tau(c_1 \otimes c_2) \cdot \alpha^\tau(r_1'{\downarrow_{\eta_2^+}}{\downarrow_{c_2}} \mathbin{\bar{\|}} r_2'{\downarrow_{\eta_1^+}}{\downarrow_{c_1}}) \\
&\quad [\text{by Inductive Hypothesis}] \\
&\leq (\tau(\eta_1^+ \otimes \eta_2^+), \bar{\tau}(\eta_1^- \uplus \eta_2^-)) \rightarrowtail \tau(c_1 \otimes c_2) \cdot \alpha^\tau(r_1'{\downarrow_{\eta_2^+}}{\downarrow_{c_2}}) \mathbin{\hat{\|}} \alpha^\tau(r_2'{\downarrow_{\eta_1^+}}{\downarrow_{c_1}}) \\
&\quad [\text{by the abstract domain properties}] \\
&\leq ((\tau(\eta_1^+) \hat{\otimes} \tau(\eta_2^+), \bar{\tau}(\eta_1^-) \mathbin{\hat{\uplus}} \bar{\tau}(\eta_2^-)) \rightarrowtail \tau(c_1) \hat{\otimes} \tau(c_2) \cdot \alpha^\tau(r_1'{\downarrow_{\eta_2^+}}{\downarrow_{c_2}}) \mathbin{\hat{\|}} \alpha^\tau(r_2'{\downarrow_{\eta_1^+}}{\downarrow_{c_1}})) \\
&\quad [\text{by Lemma A.1}] \\
&\leq ((\tau(\eta_1^+) \hat{\otimes} \tau(\eta_2^+), \bar{\tau}(\eta_1^-) \mathbin{\hat{\uplus}} \bar{\tau}(\eta_2^-)) \rightarrowtail \tau(c_1) \hat{\otimes} \tau(c_2) \cdot \alpha^\tau(r_1'){\hat{\downarrow}_{\tau(\eta_2^+)}}{\hat{\downarrow}_{\tau(c_2)}} \mathbin{\hat{\|}} \alpha^\tau(r_2'){\hat{\downarrow}_{\tau(\eta_1^+)}}{\hat{\downarrow}_{\tau(c_1)}}) \\
&\quad [\text{by Definition 3.13 and since } \tau(c_1) \hat{\otimes} \tau(c_2) \neq \hat{false}] \\
&= (\tau(\eta_1^+), \bar{\tau}(\eta_1^-)) \rightarrowtail \tau(c_1) \cdot \alpha^\tau(r_1'){\hat{\downarrow}_{\tau(\eta_2^+)}}{\hat{\downarrow}_{\tau(c_2)}} \mathbin{\hat{\|}} (\tau(\eta_2^+), \bar{\tau}(\eta_2^-)) \rightarrowtail \tau(c_2) \cdot \alpha^\tau(r_2'){\hat{\downarrow}_{\tau(\eta_1^+)}}{\hat{\downarrow}_{\tau(c_1)}} \\
&= \alpha^\tau(r_1) \mathbin{\hat{\|}} \alpha^\tau(r_2)
\end{aligned}$$

[11] We have omitted some technical details of the concrete domain, but operators $\uplus$ and conjunction (written $\otimes$) of two concrete conditions are defined similarly to the corresponding operators in the abstract domain.

$c_1 \otimes c_2 = false$ From the insertion between the concrete and abstract domains, it follows that $\tau(c_1)\hat{\otimes}\tau(c_2) = \hat{false}$.

$$
\begin{aligned}
&\alpha^\tau(r_1 \mathbin{\bar{\|}} r_2) = \\
&\quad = \alpha^\tau((\eta_1 \otimes \eta_2) \rightarrowtail false \cdot \boxtimes) \\
&\qquad [\text{by Definition 3.8 } (\alpha^\tau)] \\
&\quad = (\tau(\eta_1^+ \otimes \eta_2^+), \bar{\tau}(\eta_1^- \uplus \eta_2^-)) \rightarrowtail \tau(false) \cdot \boxtimes \\
&\qquad [\text{by the abstract domain properties }] \\
&\quad \leqslant (\tau(\eta_1^+) \mathbin{\hat{\otimes}} \tau(\eta_2^+), \bar{\tau}(\eta_1^-) \mathbin{\hat{\uplus}} \bar{\tau}(\eta_2^-)) \rightarrowtail \tau(c_1) \mathbin{\hat{\otimes}} \tau(c_2) \cdot \boxtimes \\
&\qquad [\text{by Definition 3.13}] \\
&\quad = (\tau(\eta_1^+), \bar{\tau}(\eta_1^-)) \rightarrowtail \tau(c_1) \cdot \alpha^\tau(r'_A) \mathbin{\hat{\|}} (\tau(\eta_2^+), \bar{\tau}(\eta_2^-)) \rightarrowtail \tau(c_2) \cdot \alpha^\tau(r'_B) \\
&\qquad [\text{since } \tau(c_1) \mathbin{\hat{\otimes}} \tau(c_2) = \hat{false}] \\
&\quad = \alpha^\tau(r_1) \mathbin{\hat{\|}} \alpha^\tau(r_2)
\end{aligned}
$$

$r_1 = \eta_1 \rightarrowtail c_1 \cdot r'_1$ and $r_2 = stutt(\eta_2^-) \cdot r'_2$ In case $(\eta_1^+, \eta_1^- \uplus \eta_2^-)$ is not a consistent condition $r_1 \mathbin{\bar{\|}} r_2$ is not defined and, as a consequence, $\alpha^\tau(r_1 \mathbin{\bar{\|}} r_2)$ is not defined. Otherwise, if $(\eta_1^+, \eta_1^- \uplus \eta_2^-)$ is a consistent condition, by the properties of the abstract domain and Equation 3.2, its abstraction is consistent too, thus

$$
\begin{aligned}
&\alpha^\tau(r_1 \mathbin{\bar{\|}} r_2) = \\
&\quad = \alpha^\tau((\eta_1^+, \eta_1^- \uplus \eta_2^-) \rightarrowtail c_1 \cdot r'_1 \mathbin{\bar{\|}} r'_2{\downarrow_{\eta_1^+}}{\downarrow_{c_1}}) \\
&\qquad [\text{by Definition 3.8 } (\alpha^\tau) \text{ and property (3.2)}] \\
&\quad = (\tau(\eta_1^+), \bar{\tau}(\eta_1^- \uplus \eta_2^-)) \rightarrowtail \tau(c_1) \cdot \alpha^\tau(r'_1 \mathbin{\bar{\|}} r'_2{\downarrow_{\eta_1^+}}{\downarrow_{c_1}}) \\
&\qquad [\text{by Inductive Hypothesis}] \\
&\quad \leqslant (\tau(\eta_1^+), \bar{\tau}(\eta_1^- \uplus \eta_2^-)) \rightarrowtail \tau(c_1) \cdot (\alpha^\tau(r'_1) \mathbin{\hat{\|}} \alpha^\tau(r'_2{\downarrow_{\eta_1^+}}{\downarrow_{c_1}})) \\
&\qquad [\text{by Lemma A.1}] \\
&\quad \leqslant (\tau(\eta_1^+), \bar{\tau}(\eta_1^- \uplus \eta_2^-)) \rightarrowtail \tau(c_1) \cdot (\alpha^\tau(r'_1) \mathbin{\hat{\|}} \alpha^\tau(r'_2){\hat{\downarrow}_{\tau(\eta_1^+)}}{\hat{\downarrow}_{\tau(c_1)}}) \\
&\quad = (\tau(\eta_1^+), \bar{\tau}(\eta_1^-) \mathbin{\hat{\uplus}} \bar{\tau}(\eta_2^-)) \rightarrowtail \tau(c_1) \cdot (\alpha^\tau(r'_1) \mathbin{\hat{\|}} \alpha^\tau(r'_2){\hat{\downarrow}_{\tau(\eta_1)}}{\hat{\downarrow}_{\tau(c_1)}}) \\
&\qquad [\text{by Definition 3.13}] \\
&\quad = (\tau(\eta_1^+), \bar{\tau}(\eta_1^-)) \rightarrowtail \tau(c_1) \cdot \alpha^\tau(r'_1) \mathbin{\hat{\|}} stutt(\bar{\tau}(\eta_2^-)) \cdot \alpha^\tau(r'_2){\hat{\downarrow}_{\tau(\eta_1)}}{\hat{\downarrow}_{\tau(c_1)}} \\
&\quad = \alpha^\tau(r_1) \mathbin{\hat{\|}} \alpha^\tau(r_2)
\end{aligned}
$$

$r_1 = stutt(\eta_1^-) \cdot r'_1$ and $r_2 = stutt(\eta_2^-) \cdot r'_2$

$$
\begin{aligned}
&\alpha^\tau(r_1 \mathbin{\bar{\|}} s_2) = \\
&\quad = \alpha^\tau(stutt(\eta_1^- \uplus \eta_2^-) \cdot r'_1 \mathbin{\bar{\|}} r'_2) \\
&\qquad [\text{by Definition 3.8 } (\alpha^\tau)] \\
&\quad = stutt(\bar{\tau}(\eta_1^- \uplus \eta_2^-)) \cdot \alpha^\tau(r'_1 \mathbin{\bar{\|}} r'_2) \\
&\quad = stutt(\bar{\tau}(\eta_1^-) \mathbin{\hat{\uplus}} \bar{\tau}(\eta_2^-)) \cdot \alpha^\tau(r'_1 \mathbin{\bar{\|}} r'_2) \\
&\qquad [\text{by Inductive Hypothesis}] \\
&\quad \leqslant stutt(\bar{\tau}(\eta_1^-) \mathbin{\hat{\uplus}} \bar{\tau}(\eta_2^-)) \cdot \alpha^\tau(r'_1) \mathbin{\hat{\|}} \alpha^\tau(r'_2) \\
&\qquad [\text{by Definition 3.13}] \\
&\quad = stutt(\bar{\tau}(\eta_1^-)) \cdot \alpha^\tau(r'_1) \mathbin{\hat{\|}} stutt(\bar{\tau}(\eta_1^-)) \cdot \alpha^\tau(r'_2) \\
&\quad = \alpha^\tau(r_1) \mathbin{\hat{\|}} \alpha^\tau(r_2)
\end{aligned}
$$

□

The abstract hiding operator $\hat{\exists}$ is sound w.r.t. its concrete counterpart.

Lemma A.6 *Given* $r \in \mathbf{CT}$ *and* $V \in \wp(\mathit{Var})$, $\alpha^\tau(\bar{\exists}_V\, r) \leq \hat{\exists}_V\, \alpha^\tau(r)$.

Proof. Let us first recall the definition for the concrete operator, technically adapted from [CTV13].

Definition A.7 (Concrete hiding operator) *The* concrete hiding operator *is the partial function* $\bar{\exists}\colon \wp(\mathit{Var}) \times \mathbf{CT} \to \mathbf{CT}$ *defined by structural induction as:* $\bar{\exists}_V\, \epsilon := \epsilon$, $\bar{\exists}_V\, \boxtimes := \boxtimes$,

$$\bar{\exists}_V \left((\eta^+, \eta^-) \rightarrowtail c \cdot r'\right) := \left((\exists_V\, \eta^+, \exists_V\, \eta^-) \rightarrowtail \exists_V\, c\right) \cdot \bar{\exists}_V\, r'$$

$$\bar{\exists}_V \left(\mathit{stutt}(\eta^-) \cdot r'\right) := \mathit{stutt}(\exists_V\, \eta^-) \cdot \bar{\exists}_V\, r'$$

where, for all $c \in \mathbf{C}$, $\exists_{\{x_1,\ldots,x_n\}}\, c := \exists_{x_1} \cdots \exists_{x_n}\, c$ *and, for all* $C \in \wp(\mathbf{C})$, $\exists_V\, C := \{\exists_V\, c \mid c \in C\}$.
We abuse notation and write $\exists_x\, r$ *for* $\exists_{\{x\}}\, r$.

We proceed by structural induction on r.

$r = \epsilon$ or $r = \boxtimes$ In this case the statement follows directly from Definition 3.15.

$r = (\eta^+, \eta^-) \rightarrowtail c \cdot r'$

$$\begin{aligned}
\alpha^\tau(\bar{\exists}_V\, r) &= \alpha^\tau(\bar{\exists}_V((\eta^+, \eta^-) \rightarrowtail c \cdot r')) \\
&= \alpha^\tau((\exists_V\, \eta^+, \exists_V\, \eta^-) \rightarrowtail \exists_V\, c \cdot \bar{\exists}_V\, r') \\
&\qquad [\text{by Definition 3.8}] \\
&= (\tau(\exists_V\, \eta^+), \bar{\tau}(\exists_V\, \eta^-)) \rightarrowtail \tau(\exists_V\, c) \cdot \alpha^\tau(\bar{\exists}_V\, r') \\
&\qquad [\text{by Inductive Hypothesis}] \\
&\leq (\tau(\exists_V\, \eta^+), \bar{\tau}(\exists_V\, \eta^-)) \rightarrowtail \tau(\exists_V\, c) \cdot \hat{\exists}_V\, \alpha^\tau(r') \\
&\qquad [\text{by the abstract domain properties}] \\
&\leq (\hat{\exists}_V\, \tau(\eta^+), \hat{\exists}_V\, \bar{\tau}(\eta^-)) \rightarrowtail \hat{\exists}_V\, \tau(c) \cdot \hat{\exists}_V\, \alpha^\tau(r') \\
&\qquad [\text{by Definition 3.15}] \\
&= \hat{\exists}_V\, \alpha^\tau(r)
\end{aligned}$$

$r = \mathit{stutt}(\eta^-) \cdot r'$

$$\begin{aligned}
\alpha^\tau(\bar{\exists}_V\, r) &= \alpha^\tau(\bar{\exists}_V(\mathit{stutt}(\eta^-) \cdot r')) \\
&= \alpha^\tau((\mathit{stutt}(\bar{\exists}_V\, \eta^-) \cdot \bar{\exists}_V\, r')) \\
&\qquad [\text{by Definition 3.8}] \\
&= \mathit{stutt}(\bar{\tau}(\exists_V\, \eta^-)) \cdot \alpha^\tau(\bar{\exists}_V\, r') \\
&\qquad [\text{by Inductive Hypothesis}] \\
&\leq \mathit{stutt}(\bar{\tau}(\exists_V\, \eta^-)) \cdot \hat{\exists}_V\, \alpha^\tau(r') \\
&\qquad [\text{by the abstract domain properties}] \\
&\leq \mathit{stutt}(\hat{\exists}_V\, \bar{\tau}(\eta^-)) \cdot \hat{\exists}_V\, \alpha^\tau(r') \\
&\qquad [\text{by Definition 3.15}] \\
&= \hat{\exists}_V\, \alpha^\tau(r)
\end{aligned}$$

□

It follows directly from Lemma A.6 and Equation (3.3) that, given a conditional trace $r \in \mathbb{C}$ and a variable $x \in \mathit{Var}$:

$$r \text{ is } x\text{-self-sufficient} \implies \alpha^\tau(r) \text{ is abstractly } x\text{-self-sufficient} \tag{A.1}$$

Proof of Theorem 3.19 (Correctness of the abstract semantics operators). We prove by induction on the structure of any agent A that $\alpha^\tau(\mathcal{A}[\![A]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}) \leq \mathcal{A}^\alpha[\![A]\!]_{\mathcal{I}^\alpha}$ (3.5).

Let us first recall the definition for the concrete evaluation function from [CTV13].

Definition A.8 (Semantics Evaluation Function for Agents) *Given* $A \in \mathbb{A}^{\Pi}_{\mathbf{C}}$ *and* $\mathcal{I} \in \mathbb{I}_{\Pi}$*, we define the* semantics evaluation $\mathcal{A}[\![A]\!]_{\mathcal{I}} \in \mathbb{C}$ *by structural induction as follows.*

$$\mathcal{A}[\![\mathsf{skip}]\!]_{\mathcal{I}} := \{\boxtimes\} \tag{A.2a}$$
$$\mathcal{A}[\![\mathsf{tell}(c)]\!]_{\mathcal{I}} := \{(true, \varnothing) \rightarrowtail c \cdot \boxtimes\} \tag{A.2b}$$
$$\mathcal{A}[\![A \parallel B]\!]_{\mathcal{I}} := \bigsqcup\{r_A \mathbin{\bar{\|}} r_B \mid r_A \in \mathcal{A}[\![A]\!]_{\mathcal{I}}, r_B \in \mathcal{A}[\![B]\!]_{\mathcal{I}}\} \tag{A.2c}$$
$$\mathcal{A}[\![\exists x\, A]\!]_{\mathcal{I}} := \bigsqcup\{\bar{\exists}_x r \mid r \in \mathcal{A}[\![A]\!]_{\mathcal{I}},\ r \text{ is } x\text{-self-sufficient}\} \tag{A.2d}$$
$$\mathcal{A}[\![p(\vec{x})]\!]_{\mathcal{I}} := (true, \varnothing) \rightarrowtail true \cdot \mathcal{I}(p(\vec{x}))^{12} \tag{A.2e}$$
$$\mathcal{A}[\![\sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i]\!]_{\mathcal{I}} := \bigsqcup\{\underbrace{stt \cdot \ldots \cdot stt}_{m} \cdot R \mid m \in \mathbb{N}\} \sqcup \{stt \cdot \ldots \cdot stt \cdot \ldots\} \tag{A.2f}$$

where $stt := stutt(\{c_1, \ldots, c_n\})$ *and* $R := \bigsqcup\{(c_i, \varnothing) \rightarrowtail true \cdot (r{\downarrow_{c_i}}) \mid 1 \le i \le n, r \in \mathcal{A}[\![A_i]\!]_{\mathcal{I}}\}$

$$\begin{aligned}
&\mathcal{A}[\![\mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B]\!]_{\mathcal{I}} := \\
&\quad \{(c, \varnothing) \rightarrowtail true \cdot \boxtimes \mid \boxtimes \in \mathcal{A}[\![A]\!]_{\mathcal{I}}\} \sqcup \\
&\quad \bigsqcup\{(c \otimes \eta^+, \eta^-) \rightarrowtail d \cdot (r{\downarrow_c}) \mid (\eta^+, \eta^-) \rightarrowtail d \cdot r \in \mathcal{A}[\![A]\!]_{\mathcal{I}}, \forall c^- \in \eta^-.\, c \otimes \eta^+ \nvdash c^-\} \sqcup \\
&\quad \bigsqcup\{(c, \eta^-) \rightarrowtail true \cdot (r{\downarrow_c}) \mid stutt(\eta^-) \cdot r \in \mathcal{A}[\![A]\!]_{\mathcal{I}},\ \forall c^- \in \eta^-.\, c \nvdash c^-\} \sqcup \\
&\quad \{(true, \{c\}) \rightarrowtail true \cdot \boxtimes \mid \boxtimes \in \mathcal{A}[\![B]\!]_{\mathcal{I}}\} \sqcup \\
&\quad \bigsqcup\{(\eta^+, \eta^- \uplus \{c\}) \rightarrowtail d \cdot r \mid\ (\eta^+, \eta^-) \rightarrowtail d \cdot r \in \mathcal{A}[\![B]\!]_{\mathcal{I}}, \eta^+ \nvdash c\} \sqcup \\
&\quad \bigsqcup\{(true, \eta^- \uplus \{c\}) \rightarrowtail true \cdot r \mid stutt(\eta^-) \cdot r \in \mathcal{A}[\![B]\!]_{\mathcal{I}}\}
\end{aligned} \tag{A.2g}$$

Now we proceed by cases.

$A = \mathsf{skip}$ In this case, the proof is straightforward: $\alpha^{\tau}(\mathcal{A}[\![\mathsf{skip}]\!]_{\gamma^{\tau}(\mathcal{I}^{\alpha})}) = \{\boxtimes\} = \mathcal{A}^{\alpha}[\![\mathsf{skip}]\!]_{\mathcal{I}^{\alpha}}$

$A = \mathsf{tell}(c)$

$$\begin{aligned}
\alpha^{\tau}(\mathcal{A}[\![\mathsf{tell}(c)]\!]_{\gamma^{\tau}(\mathcal{I}^{\alpha})}) &= \alpha^{\tau}(\{(true, \varnothing) \rightarrowtail c \cdot \boxtimes\}) \\
&\qquad [\text{by Definition 3.8 } (\alpha^{\tau})] \\
&= (\tau(true), \bar{\tau}(\varnothing)) \rightarrowtail \tau(c) \cdot \boxtimes \\
&\qquad [\text{since } \tau(true) = \hat{true} \text{ and } \bar{\tau}(\varnothing) = \varnothing] \\
&= (\hat{true}, \varnothing) \rightarrowtail \tau(c) \cdot \boxtimes \\
&\qquad [\text{by Equation (3.4b)}] \\
&= \mathcal{A}^{\alpha}[\![\mathsf{tell}(c)]\!]_{\mathcal{I}^{\alpha}}
\end{aligned}$$

$A = \sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i$

$$\begin{aligned}
&\alpha^{\tau}(\mathcal{A}[\![\sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i]\!]_{\gamma^{\tau}(\mathcal{I}^{\alpha})}) = \\
&\quad \alpha^{\tau}(\bigsqcup\{\underbrace{stt \cdot \ldots \cdot stt}_{m} \cdot r \mid m \in \mathbb{N}, r \in M\} \sqcup \{stt \cdot \ldots \cdot stt \cdot \ldots\}) \\
&\quad \text{where } stt := stutt(\{c_1, \ldots, c_n\}) \text{ and } M = \bigsqcup\{(c_i, \varnothing) \rightarrowtail true \cdot (r'{\downarrow_{c_i}}) \mid 1 \le i \le n,\ r' \in \mathcal{A}[\![A_i]\!]_{\gamma^{\tau}(\mathcal{I}^{\alpha})}\}. \\
&\qquad [\text{by Definition 3.8 } (\alpha^{\tau})] \\
&= \bigvee\{\underbrace{stt^{\alpha} \cdot \ldots \cdot stt^{\alpha}}_{m} \cdot \hat{r} \mid m \in \mathbb{N}, \hat{r} \in \hat{M}\} \vee \{stt^{\alpha} \cdot \ldots \cdot stt^{\alpha} \cdot \ldots\} \\
&\quad \text{where } stt^{\alpha} := stutt(\bar{\tau}(\{c_1, \ldots, c_n\}))
\end{aligned}$$

[12] Recall that we denote $\{s_1 \cdot s_2 \mid s_2 \in S\}$ by $s_1 \cdot S$.

$$
\begin{aligned}
&\text{and } \hat{M} = \bigvee\{(\tau(c_i),\varnothing) \rightarrowtail \hat{true}\cdot\alpha^\tau(r'{\downarrow_{c_i}}) \mid 1 \le i \le n,\ r' \in \mathcal{A}[\![A_i]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}\}.\\
&\qquad [\text{by Induction Hypothesis and Lemma A.1}]\\
&\le \bigvee\Big\{\underbrace{stt^\alpha\cdot\ldots\cdot stt^\alpha}_{m}\cdot\hat{r} \,\Big|\, m\in\mathbb{N}, \hat{r}\in\hat{M}'\Big\} \vee \{stt^\alpha\cdot\ldots\cdot stt^\alpha\cdot\ldots\}\\
&\quad \text{where } \hat{M}' = \bigvee\{(\tau(c_i),\varnothing) \rightarrowtail \hat{true}\cdot\hat{r}'{\hat{\downarrow}_{c_i}} \mid 1\le i\le n,\ \hat{r}'\in\mathcal{A}^\alpha[\![A_i]\!]_{\mathcal{I}^\alpha}\}.\\
&\qquad [\text{by Equation (3.4f)}]\\
&= \mathcal{A}^\alpha[\![\sum_{i=1}^{n}\mathsf{ask}(c_i)\to A_i]\!]_{\mathcal{I}^\alpha}
\end{aligned}
$$

A = now c then A_1 else A_2

$$
\begin{aligned}
&\alpha^\tau(\mathcal{A}[\![\mathsf{now}\ c\ \mathsf{then}\ A_1\ \mathsf{else}\ A_2]\!]_{\gamma^\tau(\mathcal{I}^\alpha)})\\
&= \alpha^\tau(\ \{(c,\varnothing) \rightarrowtail true\cdot\boxtimes \mid \boxtimes\in\mathcal{A}[\![A_1]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}\}\sqcup\\
&\qquad \bigsqcup\{(c\otimes\eta^+,\eta^-) \rightarrowtail d\cdot(r{\downarrow_c}) \mid (\eta^+,\eta^-)\rightarrowtail d\cdot r\in\mathcal{A}[\![A_1]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}, \forall c^-\in\eta^-.\, c\otimes\eta^+\nvdash c^-\}\sqcup\\
&\qquad \bigsqcup\{(c,\eta^-) \rightarrowtail true\cdot(r{\downarrow_c}) \mid\ stutt(\eta^-)\cdot r\in\mathcal{A}[\![A_1]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}, \forall c^-\in\eta^-.\, c\nvdash c^-\}\sqcup\\
&\qquad \{(true,\{c\}) \rightarrowtail true\cdot\boxtimes \mid \boxtimes\in\mathcal{A}[\![A_2]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}\}\sqcup\\
&\qquad \bigsqcup\{(\eta^+,\eta^-\uplus\{c\}) \rightarrowtail d\cdot r \mid (\eta^+,\eta^-)\rightarrowtail d\cdot r\in\mathcal{A}[\![A_2]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}, \eta^+\nvdash c\}\sqcup\\
&\qquad \bigsqcup\{(true,\eta^-\uplus\{c\}) \rightarrowtail true\cdot r \mid stutt(\eta^-)\cdot r\in\mathcal{A}[\![A_2]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}\})\\
&\qquad [\text{by Definition 3.8 } (\alpha^\tau)]\\
&= \{(\tau(c),\bar\tau(\varnothing)) \rightarrowtail \tau(true)\cdot\tau(\boxtimes) \mid \boxtimes\in\mathcal{A}[\![A_1]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}\}\vee\\
&\quad \bigvee\{\ (\tau(c\otimes\eta^+),\bar\tau(\eta^-)) \rightarrowtail \tau(d)\cdot\alpha^\tau(r{\downarrow_c}) \mid\\
&\qquad (\eta^+,\eta^-)\rightarrowtail d\cdot r\in\mathcal{A}[\![A_1]\!]_{\tilde\gamma(\mathcal{I}^\alpha)}, \forall c^-\in\eta^-.\, c\otimes\eta^+\nvdash c^-\}\vee\\
&\quad \bigvee\{(\tau(c),\bar\tau(\eta^-)) \rightarrowtail \tau(true)\cdot\alpha^\tau(r{\downarrow_c}) \mid\ stutt(\eta^-)\cdot r\in\mathcal{A}[\![A_1]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}, \forall c^-\in\eta^-.\, c\nvdash c^-\}\vee\\
&\quad \{(\tau(true),\bar\tau(\{c\})) \rightarrowtail \tau(true)\cdot\tau(\boxtimes) \mid \boxtimes\in\mathcal{A}[\![A_2]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}\}\vee\\
&\quad \bigvee\{(\tau(\eta^+),\bar\tau(\eta^-\uplus\{c\})) \rightarrowtail \tau(d)\cdot\alpha^\tau(r) \mid (\eta^+,\eta^-)\rightarrowtail d\cdot r\in\mathcal{A}[\![A_2]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}, c\nvdash\eta^+\}\vee\\
&\quad \bigvee\{(\tau(true),\bar\tau(\eta^-\uplus\{c\})) \rightarrowtail \tau(true)\cdot\alpha^\tau(r) \mid stutt(\eta^-)\cdot r\in\mathcal{A}[\![A_2]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}\}\\
&\qquad [\text{by Lemma A.1}]\\
&\le \{(\tau(c),\varnothing) \rightarrowtail \hat{true}\cdot\boxtimes \mid \boxtimes\in\mathcal{A}[\![A_1]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}\}\vee\\
&\quad \bigvee\{\ (\tau(c\otimes\eta^+),\bar\tau(\eta^-)) \rightarrowtail \tau(d)\cdot\alpha^\tau(r){\hat{\downarrow}_{\tau(c)}} \mid\\
&\qquad (\eta^+,\eta^-)\rightarrowtail d\cdot r\in\mathcal{A}[\![A_1]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}, \forall c^-\in\eta^-.\, c\otimes\eta^+\nvdash c^-\}\vee\\
&\quad \bigvee\{(\tau(c),\bar\tau(\eta^-)) \rightarrowtail \hat{true}\cdot\alpha^\tau(r){\hat{\downarrow}_{\tau(c)}} \mid\ stutt(\eta^-)\cdot r\in\mathcal{A}[\![A_1]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}, \forall c^-\in\eta^-.\, c\nvdash c^-\}\vee\\
&\quad \{(\hat{true},\bar\tau(\{c\})) \rightarrowtail \hat{true}\cdot\boxtimes \mid \boxtimes\in\mathcal{A}[\![A_2]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}\}\vee\\
&\quad \bigvee\{(\tau(\eta^+),\bar\tau(\eta^-\uplus\{c\})) \rightarrowtail \tau(d)\cdot\alpha^\tau(r) \mid (\eta^+,\eta^-)\rightarrowtail d\cdot r\in\mathcal{A}[\![A_2]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}, c\nvdash\eta^+\}\vee\\
&\quad \bigvee\{(\hat{true},\bar\tau(\eta^-\uplus\{c\})) \rightarrowtail \hat{true}\cdot\alpha^\tau(r) \mid stutt(\eta^-)\cdot r\in\mathcal{A}[\![A_2]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}\}\\
&\qquad [\text{by Induction Hypothesis and the abstract domain properties}]
\end{aligned}
$$

$$
\begin{aligned}
&\leq \{(\tau(c),\varnothing) \rightarrowtail \hat{true}\cdot\boxtimes \mid \boxtimes \in \mathcal{A}^\alpha[\![A_1]\!]_{\mathcal{I}^\alpha}\} \vee \\
&\bigvee\{ (\tau(c) \mathbin{\hat{\otimes}} \tau(\eta^+), \overline{\tau}(\eta^-)) \rightarrowtail \tau(d)\cdot(\hat{r}\hat{\downarrow}_{\tau(c)}) \mid \\
&\qquad (\tau(\eta^+),\overline{\tau}(\eta^-)) \rightarrowtail \tau(d)\cdot\hat{r} \in \mathcal{A}^\alpha[\![A_1]\!]_{\mathcal{I}^\alpha}, \forall c^- \in \overline{\tau}(\eta^-).\ \tau(c) \mathbin{\hat{\otimes}} \tau(\eta^+) \not\vdash c^-\} \vee \\
&\bigvee\{(\tau(c),\overline{\tau}(\eta^-)) \rightarrowtail \hat{true}\cdot(\hat{r}\hat{\downarrow}_{\tau(c)}) \mid \ stutt(\overline{\tau}(\eta^-))\cdot\hat{r} \in \mathcal{A}^\alpha[\![A_1]\!]_{\mathcal{I}^\alpha}, \forall c^- \in \overline{\tau}(\eta^-).\ \tau(c) \not\vdash c^-\} \vee \\
&\{(\hat{true},\overline{\tau}(\{c\})) \rightarrowtail \hat{true}\cdot\boxtimes \mid \boxtimes \in \mathcal{A}^\alpha[\![A_2]\!]_{\mathcal{I}^\alpha}\} \vee \\
&\bigvee\{ (\tau(\eta^+),\tau(c) \mathbin{\hat{\uplus}} \overline{\tau}(\eta^-)) \rightarrowtail \tau(d)\cdot\hat{r} \mid \\
&\qquad (\tau(\eta^+),\overline{\tau}(\eta^-)) \rightarrowtail \tau(d)\cdot\hat{r} \in \mathcal{A}^\alpha[\![A_2]\!]_{\mathcal{I}^\alpha}, \tau(c) \not\vdash \tau(\eta^+)\} \vee \\
&\bigvee\{(\hat{true},\tau(c) \mathbin{\hat{\uplus}} \overline{\tau}(\eta^-)) \rightarrowtail \hat{true}\cdot\hat{r} \mid stutt(\overline{\tau}(\eta^-))\cdot\hat{r} \in \mathcal{A}^\alpha[\![A_2]\!]_{\mathcal{I}^\alpha}\} \\
&\quad [\text{by Equations (3.4g)}] \\
&= \mathcal{A}^\alpha[\![\mathsf{now}\ c\ \mathsf{then}\ A_1\ \mathsf{else}\ A_2]\!]_{\mathcal{I}^\alpha}
\end{aligned}
$$

$A = A_1 \parallel A_2$ This case is straightforward by Lemma A.4.

$$
\begin{aligned}
\alpha^\tau(\mathcal{A}[\![A_1 \parallel A_2]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}) &= \alpha^\tau(\bigsqcup\{r_1 \mathbin{\bar{\parallel}} r_2 \mid r_1 \in \mathcal{A}[\![A_1]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}, r_2 \in \mathcal{A}[\![A_2]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}\}) \\
&\quad [\text{by Definition 3.8 } (\alpha^\tau)] \\
&= \bigvee\{\alpha^\tau(r_1 \mathbin{\bar{\parallel}} r_2) \mid r_1 \in \mathcal{A}[\![A_1]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}, r_2 \in \mathcal{A}[\![A_2]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}\} \\
&\quad [\text{by Lemma A.4}] \\
&\leq \bigvee\{\alpha^\tau(r_1) \mathbin{\hat{\parallel}} \alpha^\tau(r_2) \mid r_1 \in \mathcal{A}[\![A_1]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}, r_2 \in \mathcal{A}[\![A_2]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}\} \\
&\quad [\text{by Inductive Hypothesis}] \\
&\leq \bigvee\{\hat{r}_1 \mathbin{\hat{\parallel}} \hat{r}_2 \mid \hat{r}_1 \in \mathcal{A}^\alpha[\![A_1]\!]_{\mathcal{I}^\alpha}, \hat{r}_2 \in \mathcal{A}^\alpha[\![A_2]\!]_{\mathcal{I}^\alpha}\} \\
&\quad [\text{by Equation (3.4c)}] \\
&= \mathcal{A}^\alpha[\![A_1 \parallel A_2]\!]_{\mathcal{I}^\alpha}
\end{aligned}
$$

$A = \exists x\, A_1$ This case is straightforward by Lemma A.6.

$$
\begin{aligned}
\alpha^\tau(\mathcal{A}[\![\exists x\, A_1]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}) &= \alpha^\tau(\bigsqcup\{\bar{\exists}_x\, r \mid r \in \mathcal{A}[\![A_1]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}, r \text{ is } x\text{-self-sufficient}\}) \\
&= \bigvee\{\alpha^\tau(\bar{\exists}_x\, r) \mid r \in \mathcal{A}[\![A_1]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}, r \text{ is } x\text{-self-sufficient}\} \\
&\quad [\text{by Lemma A.6}] \\
&\leq \bigvee\{\hat{\exists}_x\, \alpha^\tau(r) \mid r \in \mathcal{A}[\![A_1]\!]_{\tilde{\gamma}(\mathcal{I}^\alpha)}, r \text{ is } x\text{-self-sufficient}\} \\
&\quad [\text{by Inductive Hypothesis and Property (A.1)}] \\
&\leq \bigvee\{\hat{\exists}_x\, \hat{r} \mid \hat{r} \in \mathcal{A}^\alpha[\![A_1]\!]_{\mathcal{I}^\alpha}, \hat{r} \text{ is abstractly } x\text{-self-sufficient}\} \\
&\quad [\text{by (3.4d)}] \\
&= \mathcal{A}^\alpha[\![\exists x\, A_1]\!]_{\mathcal{I}^\alpha}
\end{aligned}
$$

$A = p(z)$

$$
\begin{aligned}
\alpha^\tau(\mathcal{A}[\![p(z)]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}) &= \alpha^\tau(\bigsqcup\{(true,\varnothing) \rightarrowtail true\cdot r \mid r \in \gamma^\tau(\mathcal{I}^\alpha)(p(z))\}) \\
&\quad [\text{by Definition 3.8 } (\alpha^\tau)] \\
&= \bigvee\{(\tau(true),\overline{\tau}(\varnothing)) \rightarrowtail \tau(true)\cdot\alpha^\tau(r) \mid r \in \gamma^\tau(\mathcal{I}^\alpha)(p(z))\} \\
&\quad [\text{since } \tau(true) = \hat{true},\ \overline{\tau}(\varnothing) = \varnothing \text{ and } \alpha^\tau \circ \gamma^\tau = id] \\
&= \bigvee\{(\hat{true},\varnothing) \rightarrowtail \hat{true}\cdot\hat{r} \mid \hat{r} \in \mathcal{I}^\alpha(p(z))\} \\
&\quad [\text{by (3.4e)}] \\
&= \mathcal{A}^\alpha[\![p(z)]\!]_{\mathcal{I}^\alpha}
\end{aligned}
$$

Now we prove that $\alpha^\tau(\mathcal{D}[\![D]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}(p(\overrightarrow{x}))) \leq \mathcal{D}^\alpha[\![D]\!]_{\mathcal{I}^\alpha}(p(\overrightarrow{x}))$ (3.6).

$$\alpha^\tau(\mathcal{D}[\![D]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}(p(\overrightarrow{x}))) = \quad [\text{by } \mathcal{D} \text{ definition}]$$

$$\alpha^\tau\left(\bigsqcup_{p(\overrightarrow{x}):-A\in D} \mathcal{A}[\![A]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}\right) = \quad [\text{by } \alpha^\tau \text{ additivity}]$$

$$\bigvee_{p(\overrightarrow{x}):-A\in D} \alpha^\tau(\mathcal{A}[\![A]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}) \leq \quad [\text{since } \alpha^\tau(\mathcal{A}[\![A]\!]_{\gamma^\tau(\mathcal{I}^\alpha)}) \leq \mathcal{A}^\alpha[\![A]\!]_{\mathcal{I}^\alpha}]$$

$$\bigvee_{p(\overrightarrow{x}):-A\in D} \mathcal{A}^\alpha[\![A]\!]_{\mathcal{I}^\alpha} = \quad [\text{by } \mathcal{D}^\alpha \text{ definition}]$$

$$\mathcal{D}^\alpha[\![D]\!]_{\mathcal{I}^\alpha}(p(\overrightarrow{x}))$$

□